UNIVERSITY OF OVIEDO

SCHOOL OF COMPUTER SCIENCE

BACHELOR'S FINAL PROJECT

"DTT: Domain Transformation Tool"

DIRECTOR: Alberto Manuel Fernández Álvarez

AUTHOR: Raúl Estrada Ballesteros

Vº Bº del Director del Proyecto

Acknowledgements

I would like to express my sincere thanks to Alberto, the director of this project, for taking the time to guide me through this entire process and for his patience. Throughout the development of the project and the mentoring sessions I have learned a lot.

I also want to thank anyone who directly or indirectly gave me advice, believed in me, and helped me along this whole process.

Abstract

Software development aims at providing solutions for problems or necessities usually with a substantially complex and important domain. Domain-Driven Design, or DDD, is an approach where the design and implementation of the domain model are common and shared responsibilities between the development team and a group of domain experts.

This domain is so important that it's the core of the system and the rest of the application is built around it. Moreover, in software development it is quite important to keep the different concerns separated in different elements such as layers of modules. However, in practice other aspects or concerns of the application, like data persistency, might be introduced for various reasons, thus jeopardizing the design and the model.

Another common situation is to have the domain model or elements in one given form or technology. For instance, the domain might take the form of entities and relationships in a database already created and the developer may want to work with this domain but in another form such as java classes or UML models.

It is our goal to ease the developers' tasks, allowing them to code however they find most useful to them, and to work with any domain model, even if it has been expressed in a form that is not ideal for the developer. As a consequence, and due to the importance of the domain of these applications, and the common situations developers might face doing their job, the purpose of this project is two-fold: provide an easy way to remove persistency concerns introduced in the model while creating automatically the corresponding persistence configuration file, and provide developers with an easy tool to transform domain elements and concepts from one technology or form to another one.

Keywords

Domain Model, Domain Driven Design, UML Model, Databases, JPA Annotations, Domain Specific Language, Code Generation, Object-Relational Mapping, Eclipse Product, Plugin.

Resumen

El objetivo del desarrollo de software es proporcionar soluciones a problemas o necesidades, normalmente con un dominio considerablemente complejo e importante. El diseño guiado por el dominio (en inglés, *Domain-Driven Design* o *DDD*), es un enfoque en el que el diseño e implementación del modelo de dominio de las aplicaciones son responsabilidad común y compartida entre el equipo técnico de desarrollo y los expertos en el dominio del negocio.

Este modelo de dominio es tan importante que generalmente constituye el centro del sistema, y el resto de la aplicación se construye alrededor. Otro pilar fundamental del desarrollo de software es la separación de responsabilidades, de forma que distintos elementos del software como capas o módulos tengan funciones y responsabilidades únicas. Sin embargo, en la práctica esto no siempre se tiene presente y otros aspectos de la aplicación como la persistencia de datos se introducen en elementos como el modelo de dominio, contaminando el modelo y deteriorando su calidad y mantenibilidad.

Otra situación común es tener el modelo en una determinada tecnología o forma que no es la adecuada para el trabajo actual. Por ejemplo, un equipo de trabajo puede tener que usar el modelo de una base de datos ya existente, con sus tablas y relaciones. Sin embargo, el desarrollador puede querer este modelo en clases java o en un fichero de modelo UML.

Por tanto, el objetivo de este proyecto es simplificar y ayudar en el mayor grado posible el trabajo de los desarrolladores, permitiéndoles trabajar con modelos en distintos formatos realizando las conversiones automáticamente. Éstos también podrán codificar de manera que les sea más sencilla y productiva, como introduciendo aspectos de persistencia en el modelo con anotaciones, ya que la herramienta posteriormente se encargará de separar las responsabilidades y limpiar el código de estas configuraciones de persistencia.

Palabras clave

Modelo de dominio, Diseño guiado por dominio, modelo UML, bases de datos, anotaciones JPA, Languajes específicos del dominio, Generación de código, Mapeo Objeto-Relacional, Producto de Eclipse, Plugin.

Table of Contents

1. Introduction

1.1. Project justification

Software development must often deal with complex domains where the model becomes the core of the application. Services and other system components are built around it.

One of the pillars of software design is to design components or software elements that have only one concern or responsibility, or a set of tightly connected responsibilities. High level concerns or responsibilities usually include presentation, business logic, domain model and persistency. However, when building and coding the software elements that represent the domain, developers might decide to introduce persistency concerns in the model since it's easier and more familiar to them. While this increases their productivity and understanding of the data persistency configuration, it damages the software design and compromises the domain model.

The system becomes harder to maintain since developers cannot pinpoint a persistency error to an element in the persistency layer. The domain model cannot be reused in another application since data persistency configuration has been inserted in the model, and the other application might have or need a different configuration. There are plenty of reasons like the previous ones that illustrate the many problems that arise from introducing persistency concerns in the domain model.

Another common situation many developers face in their everyday activities is the requisite to work with a domain whose model is in a form unsuitable for the task in hand. Maybe because the domain model is a database that already exists and has been handed to them, sometimes called legacy because the developer inherits the database, and must work with the existing model, expressed in the form of entities and relationships. In this example, the developer may want to create the software elements that represent such domain, the persistency configuration file or a UML model from the database. Analyzing the metadata of the database to understand the domain model and perform the transformations to create another form of the model would take some time that the developer can better spend on something else.

What's more, the domain experts might be the ones designing and creating the domain. Since they do not have to be technical people and may not have programming skills, the tool offers a custom domain specific language close to the natural language that experts can use to express these models. Moreover, the tool provides an editor that helps express the domain with autocomplete options, checks and validations. The project also arises to provide a way to help experts and developers come together and integrate the work of the former to the work of the latter. Succinctly, developers may take the domain designed by the experts and transform the model elements into different forms that are more suitable for them, like a script to create the database with such domain, a UML model, the persistency configuration file or the java clean code.

Like the previous ones, more scenarios may arise and subsequent transformations may be needed by developers.

The project consists of different components to be used by developers to ease their job and help them work in a way that is more suitable for them, increasing their productivity while maintaining good code quality and design. It may also be used by domain experts that do not have technical skills, to be part of the team. This follows one of the main ideas behind Domain Driven Design: design and implementation of the domain is a shared responsibility between the development team and the domain experts. All of them take authorship of the model. It also helps implement the domain easily and productively, and the tool will be used to help increase code quality and good practices.

The tool is composed of different components so that it can be used in various ways. Therefore, the development team can choose whichever is better for them.

These different products include a command-line program that can be executed in a plain command-line terminal, an Eclipse plugin that can be installed in the Eclipse IDE (Integrated Development Environment, a program developers use to write code), a Maven plugin that can be used in any project where Maven is used and therefore can be executed when the project is built, installed or deployed, for instance. Finally, regarding the Domain Specific Language, an additional Eclipse plugin is offered, as well as an Eclipse product that will open an Eclipse workbench instance with the DSL editor already installed in it.

Figure 1 shows a diagram representing the high level products and their relationships.

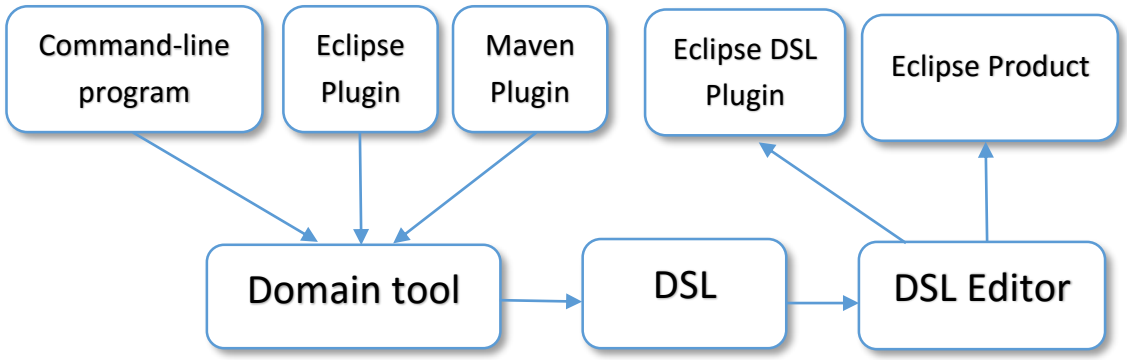

Figure 1. Overview of the different products of the project

In the end, the tool can read and analyze the domain model stored in a database, expressed as .class files (files with source code compiled by the java compiler), as .java files (files with the source code), as a UML model or expressed using the Domain Specific Language. Either way, the tool allows developers to transform the model elements and create the XML persistency configuration file (ORM.xml, Object-Relational Mapping), clean java source code, a UML model or scripts to create the different entities and relationships in a database. The different options can be seen in Figure 2.

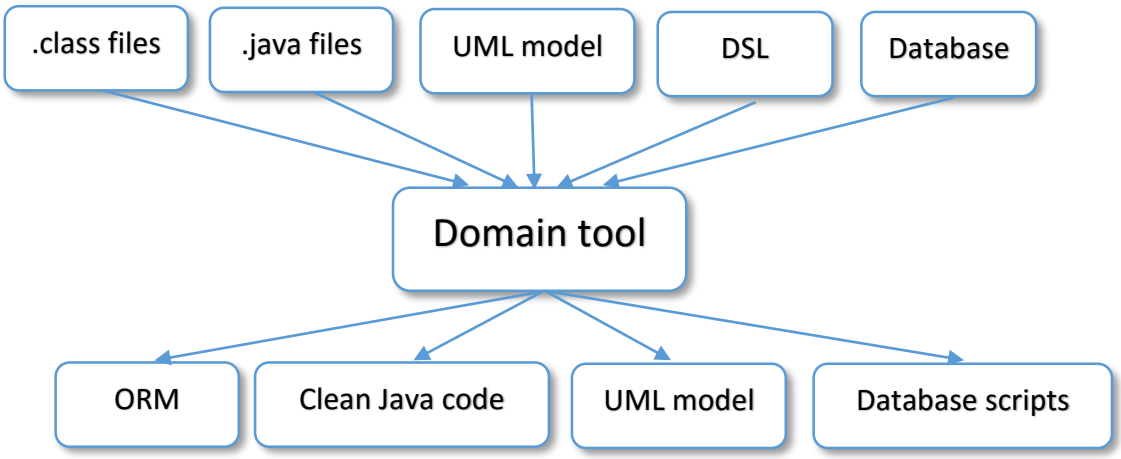

Figure 2. Transformations offered by the domain tool

1.1.1. Motivation example

The motivation of this project is to create a tool that allows developers to keep using annotations in their source code, and it automatically generates de XML configuration file for them, removing the annotations and cleaning the domain model of persistence concerns. This tool will combine both approaches, getting the best of them both, providing their benefits and avoiding their drawbacks. The automatic generation of the configuration file will be transparent to the programmer, whose only task will be to execute the proposed tool on their application.

Once the configuration file has been created and the annotations have been removed with no additional effort from the development team, the domain won't suffer from low cohesion, will be cleaner and reusable, since it can be used in other contexts that may persist different information from this model in a different way or in a different information repository.

Changes on the mapping after the XML document has been created should be made to this file. No recompilation or redeployment processes are necessary, the application doesn't need to be stopped, denying services to users and potential customers that may result in financial losses, since changes to this configuration file take effect immediately.

To illustrate this the following example is provided.

A development team is working on a management application that allows the human resources department of the business to have a better control of their managers and employees. The team has realized their domain and business rules are complex enough to apply DDD, and they have been able to integrate domain experts in their team. The first thing they did was to create the shared *Ubiquitous* language. Then, they discussed and modeled the domain and business rules into a domain model. However, when it was time to implement and code the domain, the team still had a database-driven approach in their minds and added numerous annotations to the domain classes. In the Manager class shown in Figure 3, they not only annotated persistence configuration such as what information should be persisted, format, constraints, indices of the table and generator strategies and elements of the

database, but they also introduced another concern in the domain class: Queries to retrieve specific data from the database. The java class looked something like the following.

```
@Entity
@Table(name = "t_manager", indexes = {
  @Index(columnList = "manager_department_id"),
  @Index(columnList = "manager_id")
  }, uniqueConstraints = {
    @UniqueConstraint(columnNames = {"manager_identifier"})
  })
@NamedQueries({
  @NamedQuery(name = "Manager.findByIdentifier",
  query = "SELECT m FROM Manager m WHERE m.managerIdentifier = :empId"),
  @NamedQuery(name = "Manager.findByDepartment",
  query = "SELECT m FROM Manager m WHERE m.department.id = :dptId")
})
public class Manager implements Serializable {
  private static final long serialVersionUID = 1L;

  @Id
  @Column(name = "manager_id",
     columnDefinition = "ID element used for the database primary key")
  @GeneratedValue(strategy = GenerationType.SEQUENCE,
     generator = "manager_seq_id")
  @SequenceGenerator(name = "manager_seq_id", allocationSize = 30,
      initialValue = 10, sequenceName = "manager_seq_id")
  private Long id;

  @Version
  @Column(name = "manager_ol", nullable = false,
     columnDefinition = "Version attribute to know when the entity was last
modified")
  private Long version;

  @Column(name = "manager_name", nullable = false, columnDefinition = "Full
name with first and last name of the manager")
  private String name;

  @Column(name = "manager_date_of_birth", nullable = true)
  @Temporal(TemporalType.DATE)
  private Date dateOfBirth;

  @Column(name = "manager_identifier", nullable = false,
    columnDefinition = "Unique identifier the company assigns to each
manager for their electronic card and records")
  private String managerIdentifier;

  @ManyToOne(fetch = FetchType.LAZY)
  @JoinColumn(name = "manager_department_id", nullable = false)
  private Department department;

  @OneToMany(cascade = {CascadeType.ALL}, mappedBy = "manager", fetch =
FetchType.LAZY, orphanRemoval = true)
  @OrderBy("nif DESC")
  private List<Employee> employees;

  @Enumerated(EnumType.STRING)
  @Column(name = "manager_position", nullable = false)
  private ManagerPosition position;

  @Transient
  private Integer numberOfBilingualEmployees;

     //Here there would be different methods and getters/setters
     …

}
```

Figure 3. Example of the Manager domain class with annotations

Figure 3 shows the Manager class with persistence annotations. The class itself was designed to be fairly simple and contain few attributes, but the team filled the code with annotations because it was easier for them to use annotations. They decided to include the queries in the source code as well so that queries that are related to this entity would be defined on top of the entity as well. The team wasn't confident enough to use the configuration XML file, they wanted to detect syntactic errors with annotations, they already used annotations for other purposes and it was more comfortable and easier to make sense of the configuration in smaller portions next to the element they were configuring.

However, now it's harder for the domain experts to understand and read the class, and a brief explanation from the developers won't suffice since domain experts see persistence configuration but they don't know anything about persistency, queries or database schemas.

Moreover, the company was planning on reusing this domain model for other parts of the system with other persistence models and different mappers. Another system was going to use this class, but table and column configurations are not appropriate, some attributes are not going to be persisted, and the number of bilingual employees, that is mapped as *@Transient* and therefore is not going to be persisted, needs to be stored in that different scenario.

The company already deployed the application with the Manager class as shown in Figure 3, and now they need to change some of the configuration properties. In order to do this, they will need to send an email to all the users of this application so that they don't use the application while they stop it, modify the annotations and recompile and redeploy. Time will be unnecessarily wasted.

If the tool proposed in this project is used, developers can still code the annotations in the source code, enjoying the previously commented benefits of doing so. Then, the tool is executed on the domain model classes and the XML configuration file is generated automatically and in a transparent way to the developer. The XML file created contains the configuration of all the model classes. In this example, the tool is executed only on the Manager class and the output is the following.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings version="2.1"
xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd">
 <named-query name="Manager.findByIdentifier">
   <query>SELECT m FROM Manager m WHERE m.managerIdentifier = :empId</query>
 </named-query>
 <named-query name="Manager.findByDepartment">
   <query>SELECT m FROM Manager m WHERE m.department.id = :dptId</query>
 </named-query>
 <entity class="xml.Manager">
   <table name="t_manager">
     <unique-constraint>
       <column-name>manager_identifier</column-name>
     </unique-constraint>
     <index column-list="manager_department_id"></index>
     <index column-list="manager_id"></index>
   </table>
   <attributes>
     <id name="id">
       <column name="manager_id" column-definition="ID element used for the
database primary key"/>
       <generated-value strategy="SEQUENCE" generator="manager_seq_id" />
       <sequence-generator name="manager_seq_id" allocation-size="30"
initial-value="10" sequence-name="manager_seq_id">
       </sequence-generator>
     </id>
     <basic name="name">
       <column name="manager_name" nullable="false" column-definition="Full
name with first and last name of the manager" />
     </basic>
     <basic name="dateOfBirth">
       <column name="manager_date_of_birth" nullable="true" />
       <temporal>DATE</temporal>
     </basic>
     <basic name="managerIdentifier">
       <column name="manager_identifier" nullable="false" column-
definition="Unique identifier the company assigns to each manager for their
electronic card and records"/>
     </basic>
     <basic name="position">
       <column name="manager_position" nullable="false"/>
       <enumerated>STRING</enumerated>
     </basic>
     <version name="version">
        <column name="manager_ol" nullable="false" column-
definition="Version attribute to know when the entity was last modified"/>
     </version>
     <many-to-one name="department" fetch="LAZY" target-
entity="xml.Department">
         <join-column name="manager_department_id" nullable="false"/>
      </many-to-one>
      <one-to-many name="employees" fetch="LAZY" target-
entity="xml.Employee" mapped-by="manager" orphan-removal="true">
        <order-by>name DESC</order-by>
        <cascade>
          <cascade-all />
        </cascade>
      </one-to-many>
      <transient name="numberOfBilingualEmployees" />
    </attributes>
  </entity>

</entity-mappings>
```

Figure 4. Example of the XML Configuration file

The XML validator rejects the configuration file if it doesn't follow the structure required for this document, and the order of its elements. When using the tool, developers don't have to worry about this.

This tool not only generates the previous document, but it also removes the annotations from the domain model code. The final Manager class is as follows.

```
public class Manager implements Serializable {
   private static final long serialVersionUID = 1L;

   private Long id;
   private Long version;
   private String name;
   private Date dateOfBirth;
   private String managerIdentifier;
   private Department department;

   private List<Employee> employees;
   private ManagerPosition position;
   private Integer numberOfBilingualEmployees;

      //Here there would be different methods and getters/setters
      …
}
```

Figure 5. Example of the Manager domain class without annotations

The class in Figure 5 is readable. Domain experts can easily understand it, and in some cases maybe a brief explanation from the development team can suffice. This class doesn't suffer from low cohesion as the one in Figure 4. It's focused only on representing the domain and business rules. Maintainability has increased as well since the persistency and model concerns are now in separate elements. Detecting and fixing an error is simpler as well, since the software elements don't have mixed concerns so developers don't have to go through big classes with lots of responsibilities. Therefore, the tool helps establish and improve the separation of concerns in the domain layer with regards of persistency.

Succinctly, developers coded in a way that best suited them best and the company can now reuse the domain model without transferring persistence configuration that only applies to this system. The configuration XML file was generated in the appropriate location, and if this configuration needs to be updated, the file can be modified directly and the changes will take effect immediately without having to schedule any downtime that makes the system unavailable to users.

1.2. Goals and objectives

After highlighting the importance of the domain models and briefly presenting the different problems developers may face when working with such domain models, the aim and purpose of the domain tool, as the product of this project, have been established. The goals and objectives are as follows:

- Provide an easy and efficient way to express the essence of a domain in different ways by different people.
- Allow developers to implement the domain with persistency concerns, then cleaning the code of these concerns and generate automatically the XML configuration file.

- Facilitate the transformation and creation of a domain model from legacy systems, such as UML models, annotated source code files or databases.
- Create different products of the tool to best help developers in ways that suit their work and projects best.

1.3. Analysis of the current situation

Software development aims at providing solutions for problems and necessities. In some occasions, the product of the development process must work with a substantially complex and important domain, and this is where a design approach called ***Domain-Driven Design***, or DDD, might be used. Nowadays systems are becoming more and more complex, and they take many different forms such as mobile apps, web applications, wearable apps and/or desktop programs. However, they all usually share the same domain, which remains extremely relevant. DDD advocates for a domain model design and implementation that are shared and common responsibilities between the development team and a group of domain experts.

Firstly, a shared common language, named ***Ubiquitous Language***, is established between the technical developers and the domain experts and it's very important to drive design, implementation and discussions about the domain using such language.

The domain model is a fundamental aspect of the product. Not only is the rest of the system built around it, but there is also a close connection between the domain and the source code expressing and representing it. The domain model must be a strong foundation the application relies on and builds upon.

After identifying restrictions and necessities of the product, sometimes these impose special designs or structures. This is where architectural and design patterns turn out to be helpful. Patterns are generic solutions that experience have proved to be useful under certain conditions facing certain recurrent problems. Because they're generic solutions, they must be adapted and personalized to fit the current application and domain. A well-known and widely used architectural pattern is the multilayered architecture, where usually different layers focus on different application concerns: there is usually a presentation layer, a business logic layer, a persistency layer and a domain layer that is transversal to the other ones. Each layer must focus on just one application focus. This way, the presentation layer deals with user interfaces and interaction, and how the information is displayed, but it does not know about persistency or business rules. Similarly, the domain layer contains model elements but it does not or should not have persistency concerns. There should be a clear-cut separation of responsibilities and concerns. However, in practice this is usually not the case.

Moreover, and related to the previous idea, a fundamental principle of software design is called ***high cohesion***. By designing software elements to have just one or a set of strongly connected responsibilities, developers can reap several benefits. For instance, future changes and maintainability are easily supported. To illustrate this, imagine an accounting application. This software product is made up of different components, *Clients-Bill-Retriever* being one of them. *Clients-Bill-Retriever* is in charge of retrieving and obtaining all the bills from a given client directly from the database. Furthermore, it was also designed to manage the user interface where the user selects a client via a web page to later retrieve all of his/her bills. In

this example, the software component suffers from low cohesion since it has multiple responsibilities that are somewhat, but not strongly, connected: presentation and persistency concerns. This damages readability and maintainability. This is not unusual in software development.

The domain model is no exception and it must be designed to be highly cohesive. Model objects must clearly reflect and represent the domain of the business and domain rules. Other aspects and concerns are outside their scope.

1.3.1. Domain model, persistency model

Because of the evolution of software development and trends in design, developers have acquired a database-driven focus. However, DDD requires them to change this database-driven focus when designing and implementing the domain model. In occasions, developers tend to design domain elements taking into great consideration how persistency will be implemented and configured. This way, the domain model ends up being just a data placeholder for the persistence process. Mapping the domain to the database is done almost straightforward, but the domain becomes *anemic*, that is, it just carries or holds data retrieved from a database, or holds data that will be persisted in the database, rather than representing the domain of the application and displaying the rich domain behavior it should.

Therefore, two models can be identified: the domain model and the persistence model. They should have two separate and different concerns. While the former models behavior and best represents the business domain, the latter models storage structure and configuration and deals with what information is going to be stored and where. DDD aims at creating a domain model rich in behavior that makes sense in a given context boundary and that uses the shared *Ubiquitous language* established for that domain. This model should not contain infrastructure elements and it should only depend on other possible domain models. **The domain model should be independent** from any other concerns, responsibilities or software elements. If the persistency technology changes or the database is modified for any reason, changes are only needed in the persistency model. The domain model, the core of the system, remains intact. Consequently, the impact of changes is considerably reduced and the domain model can be reused by other applications that may map data from this model to other different persistency models.

Since relational databases and object-oriented programming are both widely used in the industry, the former for persistence and the latter for implementing applications, Object-Relational Mappers, or ORMs, are needed and have become very popular. They perform transformations from the relational database to the object oriented model and vice versa, increasing productivity and performance, and reducing additional source code needed. Developers can still code using Object-Oriented approaches, and they can configure the mapping using few lines of code. Moreover, these mappers are optimized to provide good performance.

1.3.2. JPA: Annotations and XML configuration files

Even though the domain must be the core of the application and it should not be confused or mixed with persistency, persistency is key in almost every system and it must be

properly configured. When using and ORM with the Java programming language, JPA (*Java Persistence API*) is very popular. There are two common ways to configure the ORM: with an XML configuration file, and introducing annotations in the source code.

Annotations are structures that provide metadata about an element. They are placed in the source code next to the element they configure or give information about. Nowadays they are widely used for various purposes: to mark methods as obsolete or deprecated, to add information to the java documentation, to override a method and give a new implementation for it, to create tests, define the scope of a web component, etc. Developers already used annotations frequently and are familiar to them, thus making them easier to use than configuring XML files. Developers, the target users of this project, may not be familiarized or as fluent with the XML language or structure.

Writing persistence XML files can become quite complex if there's a high number of elements to configure since its contents might grow fast and a big XML file can seem chaotic and difficult to read and navigate. This may produce cognitive overload and the developer may not be able to process the information the file contains easily. However, annotations are placed with the elements they configure, therefore the developer finds smaller pieces of information that are easier to read and comprehend, expressed with structures he/she already knows about.

Moreover, the order in which annotations are placed is not important. In contrast, the content of the XML file must follow a given structure or schema previously defined in another document. Should the elements of the XML file be placed in any other order, the file will not be validated and errors difficult to pinpoint may arise.

Another important aspect of annotations is the fact that they are compiled. This helps developers detect errors early before deploying the application. For instance, when using annotations, a developer can use the @Override annotation with a method in the child class of an inheritance relationship to indicate that this implementation should be used and not the one in the parent class. If the class is not part of an inheritance relationship or the parent class doesn't have that method declared, an error would be raised. This helps developers make sure they are in fact overriding the method from the parent class at compile time. Otherwise, this error might have been difficult to detect and it might have taken the developer longer to notice it and fix it.

The compile aspect of annotations is a double-edge sword: it yields an early detection of errors, but it can also be inconvenient. Because they are compiled, every time an annotation is modified, the application must be recompiled and redeployed for changes to take effect. This produces an unnecessary waste of time. Using an XML configuration file would solve this problem since no code modification or recompilation is necessary, and changes take effect without the need to redeploy the application. This can be useful in applications where the development and maintenance processes make frequent changes to annotations or their values. This is especially useful in applications that are already deployed, need to make some modification to some mapping aspect or property configuration, and cannot stop the application. With an XML configuration file, they can make those changes without having to halt the system. Doing otherwise, recompiling and redeploying can take time, create inconveniences for the users of the application and carry economic losses for the business.

Another important drawback of annotations, and related to previous ideas commented in this section, is related to the concept of high cohesion. Annotations introduce persistence

concerns in the domain model. The code is not as clean, readable and maintainable as it could and should be. Responsibilities should be designed and assigned correctly to the appropriate elements.

1.3.3. Legacy domains

It isn't unusual for developers to have the domain model inherit from a previous team or a previous project. It can be legacy code, and the domain model classes were designed and implemented by other teams and the new project must work with those domain classes. It can also be a legacy database, designed and used by another project or a previous and different development team. It can take many different forms. Either way, these kinds of situations happen often in the industry and developers must find a way to work with them. They have to spend good time analyzing the inherited domain in whatever form it has been expressed, create and transform it into a new form that is more suitable for them (like taking the database and create the corresponding clean java code and XML configuration file).

Furthermore, the domain experts usually do not have technical skills and in many software development companies they try to explain the domain model to the development team. Some information is lost in this explanations, some other information may not be explained for various reasons, and some information will be misunderstood by the development team. Subsequently, when the development team designs and implements the model, it will not be correct. But since it has been coded and expressed with a programming language, domain experts may not understand enough to check if it's correct or not.

1.3.4. Current tools

Currently there are some tools focused on providing one or two of the necessities this project aims at meeting. However, they offer too many features for just one small part of the problem, and they can be too complex. The user may have to end up configuring many aspects he/she does not need. Tools such as MetaEdit+, Entity-DSL, jOOQ's code generator, IntelliJ IDEA's persistency mapping generator, or Eclipse UML generators may require not easy installations and integrations. Some tools are not free. Others offer their services packaged in jars to be used by other applications, thus unsuitable for the problems this project face. All of them focus on a small part of the problem, but an understanding and abstraction of the bigger problem and the solutions development teams need is necessary.

1.4. Project proposal

The project proposed a solution whose high level components can be seen in Figure 1, page 20, and the supported domain transformations are shown in Figure 2, page 21. Basically, the project proposes the creation of the following products in order to ease the work of developers:

- A command-line program.
- An Eclipse plugin.

- A Maven plugin.
- An Eclipse product with the DSL editor integrated.
- An eclipse plugin for the DSL editor.

In order to use the first three, first the user must load a model from either one of the supported input forms (a collection of class files, a collection of java files, UML models, DSL files or databases). Once loaded, the user can choose to perform as many transformations as he/she wishes to any of the supported output forms (clean java code, ORM.xml persistence configuration file, database scripts, or UML model file).

All three products are clients of another project, the core of the application, which contains the business logic to perform these services. A project was developed specifically to create the DSL, and this project and the core one were integrated so that the core could provide services related to the DSL as well. Because the DSL project provides a very useful editor with options such as autocomplete and validations, two additional products were created: An Eclipse product users can execute or launch and will open an Eclipse workbench instance with the editor already installed. The other one is an Eclipse plugin users can install in their Eclipse workbench instance.

1.4.1. Command-line program

The basic steps to use the command-line program are as follows:

- Use the ***read*** command to load the model from any of the supported forms.
- The program will display a message with the result of the loading operation.
- Use the ***write*** command to create the new model or form of the domain model loaded.

1.4.2. Eclipse plugin

Once installed, the Eclipse plugin offers its services through popup menus when clicking different elements. However, the basic work flow is the same: a model must be loaded first, then the result will be displayed to the user, and then the user can select whichever supported output form to transform the loaded model to. The popup menus appear as follows:

- Eclipse project: will appear the Load from database option.
- Eclipse package: will appear the Load from java option.
- Eclipse folder: will appear the Load from class option.
- File with .uml extension: will appear the Load from UML option.
- File with .tfg extension: will appear the Load from DSL option.
- All Transform to… options are available in each one of the them.

1.4.3. Maven plugin

The Maven plugin also follows the same basic work flow. In the pom.xml file the user must add the Maven plugin and configure where the domain model to be loaded is located, and which transformations he/she wants to perform, and where.

- Configurations go inside the ***<toolConfig>*** element.
- ***<directoryInput>*** and ***<databaseInput>*** configure the location of the input domain model, and the type of input.
- ***<toDirectory>*** configures where the output of the transformations will be placed, and ***<conversions>*** specifies all the different transformations and conversions to perform.

1.4.4. Eclipse DSL plugin and Eclipse product.

The user must add the Eclipse plugin to his/her workbench instance. The Eclipse product launches a new Eclipse instance with the plugin already installed, so the user only has to execute it.

Either way, the editor will be applied to files with the .tfg extension. The creation of these files is the same as the creation of any other file in Eclipse. These two products are created to ease the process of expressing the domain with the DSL, to provide features like autocomplete, validations, an outline or tree structure of the domain being created, etc. However, users may choose to create and write the file anyway they want. As long as it has the .tfg extension, it will be considered a DSL file.

1.5. Analysis of alternatives

In this section some other tools and products currently available to developers are analyzed to see their characteristics, how they provide a solution for developers' necessities and how they compare to this project's domain tool.

1.5.1. SQL2JAVA

SQL2JAVA is an open source tool developers can use to read the schema of a database and generate automatically the java source code classes representing the domain model. It works with many different databases that come with the JDBC driver. As a requirement, the user must have the software tool Ant installed.

The tool does not need to be installed. The user downloads a zip folder with all the necessary files, opens a command-line inside the unzipped downloaded folder and executes Ant.

The tool comes with a database connection pre-configured, but does not warn the user about it. So, when the user first executes the tool, an error like the one in Figure 6 is displayed.

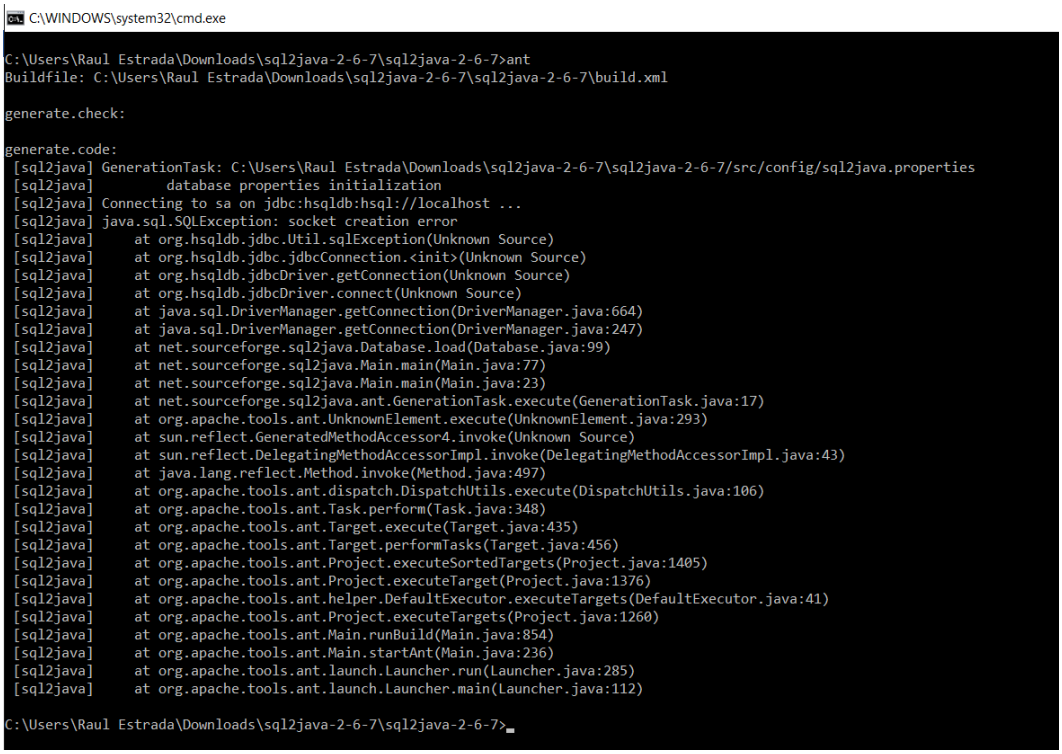

Figure 6. SQL2JAVA initial error

As the user does not learn much about the problem from the error, he/she has to look into the different configuration files and eventually change the configuration in a properties file to connect to his/her database.

After some other configurations and modifications the user must perform in order to get the tool up and working, it eventually creates the different java files with the domain model and the java documentation related to them.

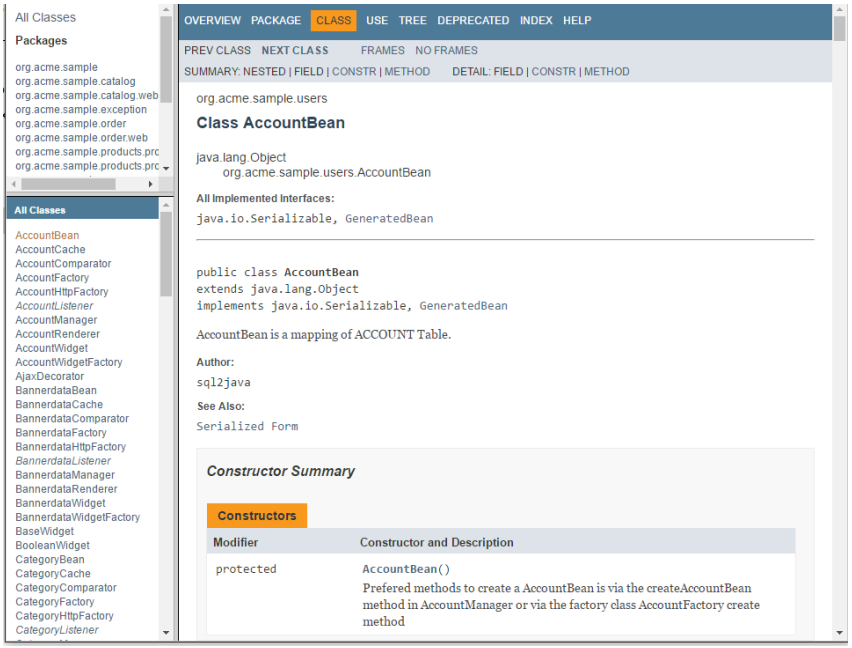

Figure 7. SQL2JAVA documentation example

With not much documentation, the tool is not very easy to use.

1.5.2. CodeTrigger Code Generator

CodeTrigger is a tool that can be integrated with Visual Studio, a software development IDE, or run as a standalone application. It analyzes the schema of SQL Server 2005+, Oracle 10g/11g or MySQL 5.5+ databases and generates C# code with the domain model classes. Users can choose to generate just model classes, or services as well.

The tool is not free, and users must register in order to download it. The official webpage offers examples and documentation on how to integrate the tool with Visual Studio and how to use the tool.

As users create the new project, CodeTrigger asks them to introduce the different configurations it needs.

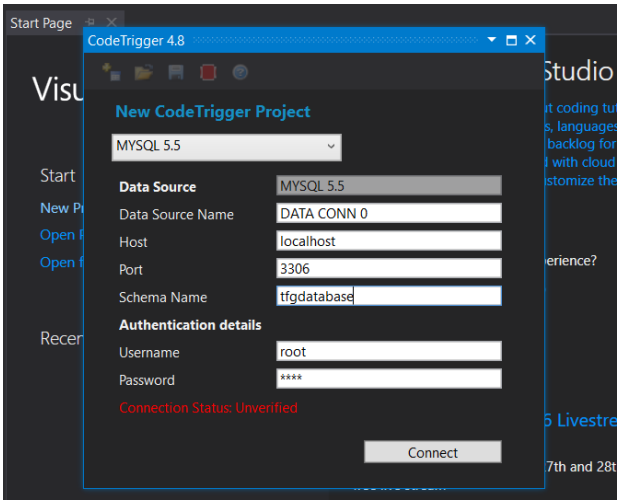

Figure 8. Database connection configuration with CodeTrigger

It generates a SQL file to be run against the database and the different C# files with the domain model classes. An example is shown in Figure 9.

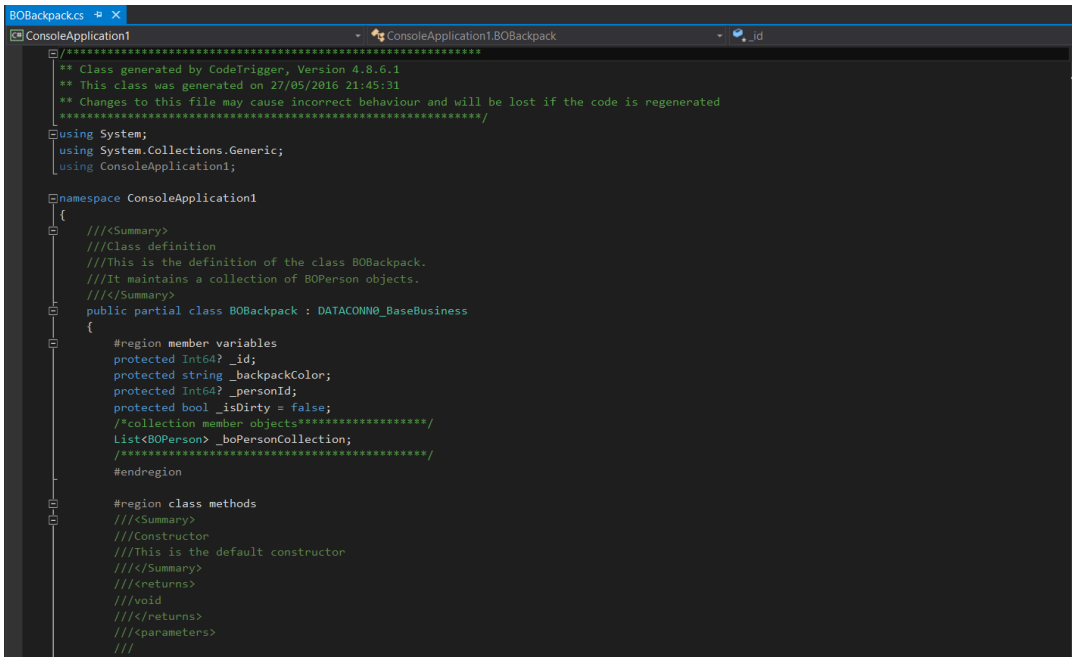

Figure 9. Domain model class auto generated by CodeTrigger

The tool offers many additional services to generate code to handle different services such as services and data access, and to generate different software elements like interfaces and factories.

1.5.3. MetaEdit+

MetaEdit+ is a software product that offers a Domain-Specific Modeling environment where users can generate domain models graphically, and then it automatically generates the code from the model. Users can create new custom generators to create code or their own DSL, code or any other output. It comes with two tools and the process workflow is as follows:

Firstly, the modeling language is designed using the MetaEdit+ Workbench. Entities, properties, value objects, custom types and rules are defined in this phase.

Secondly, the language is modeled with diagrams and generators.

The product can be integrated with software development products such as Eclipse or Visual Studio. MetaEdit+ is not free, and there's plenty of information and support available.

It offers many features and the possibility to create custom generators, and the product ends up being somewhat complex and not straightforward to use.

Figure 10 shows how entity objects are created using MetaEdit+ Workbench with a graphical user interface.

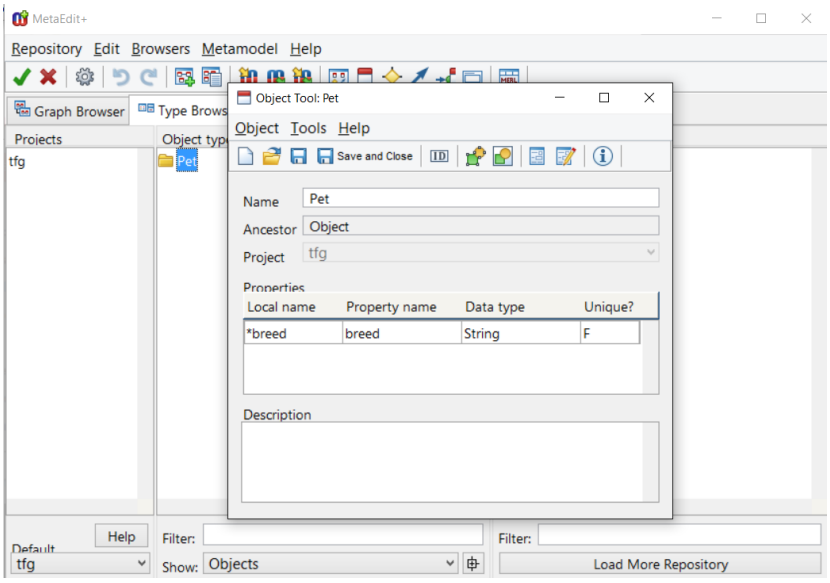

Figure 10. Designing model language using MetaEdit+ Workbench

Figure 11 shows how the domain is modeled with diagrams using the MetaEdit+ Modeler.

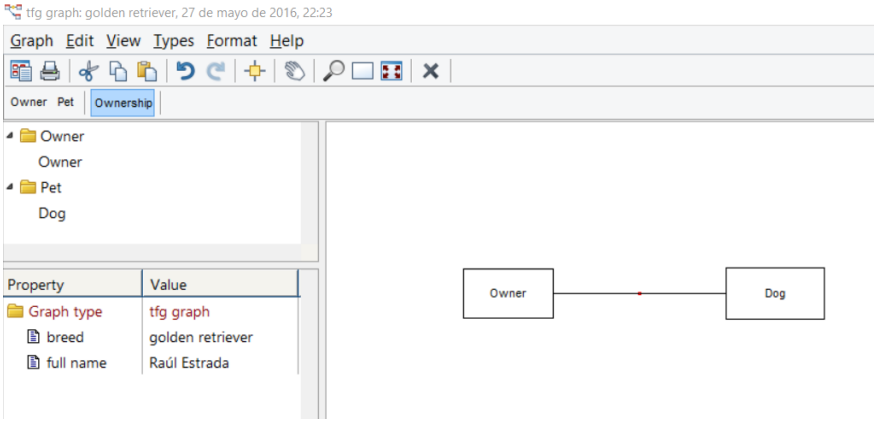

Figure 11. Modeling the language with MetaEdit+ Modeler

1.5.4. Entity-DSL

Entity-DSL is a tool that offers the possibility of expressing a domain model with a custom DSL it provides. Once the user has written the domain using the DSL, java code is automatically generated by Entity-DSL.

There isn't much support and very little documentation to aid the user install the tool, use it or fix and avoid problems.

Entity-DSL is offered for free as an Eclipse plugin that can be installed in Eclipse instances.

The DSL allows users to create quite complex and complete domains, supporting java documentation inserted in the code, and implementations for the operations. This can be a double-edge sword: it's very powerful and flexible, but also quite complex. Domain experts,

which do not have technical and programming skills, may not find it easy to use and the tool will fail to help the development team follow DDD, which specifies that the domain design and implementation is a shared responsibility between domain experts and developers.

The user can choose to compile the DSL to plain java objects, or generate JPA entities with persistency annotations. Figure 12 is an image taken from the Entity-DSL official webpage[1] and shows an example of a DSL file with the domain model.

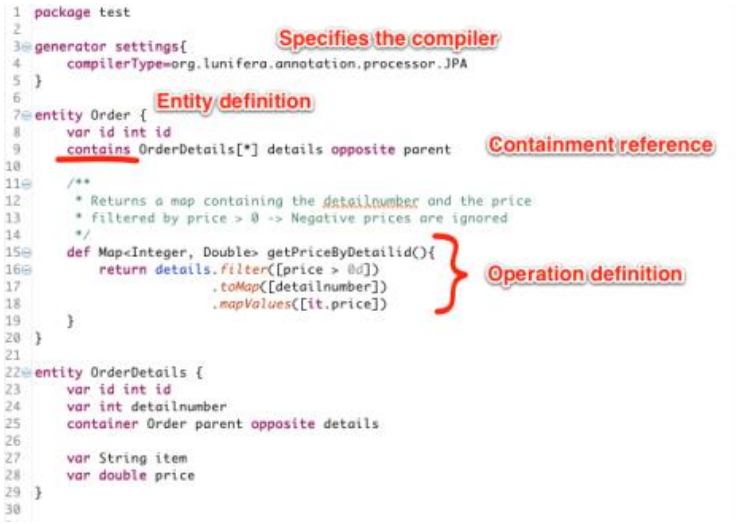

Figure 12. Entity-DSL file example.

Figure 13 is an image taken from the Entity-DSL official webpage and shows the output of the tool, a JPA entity expressing the domain model with persistence annotations.

[1] https://lunifera.wordpress.com/dsls/entitymodel/

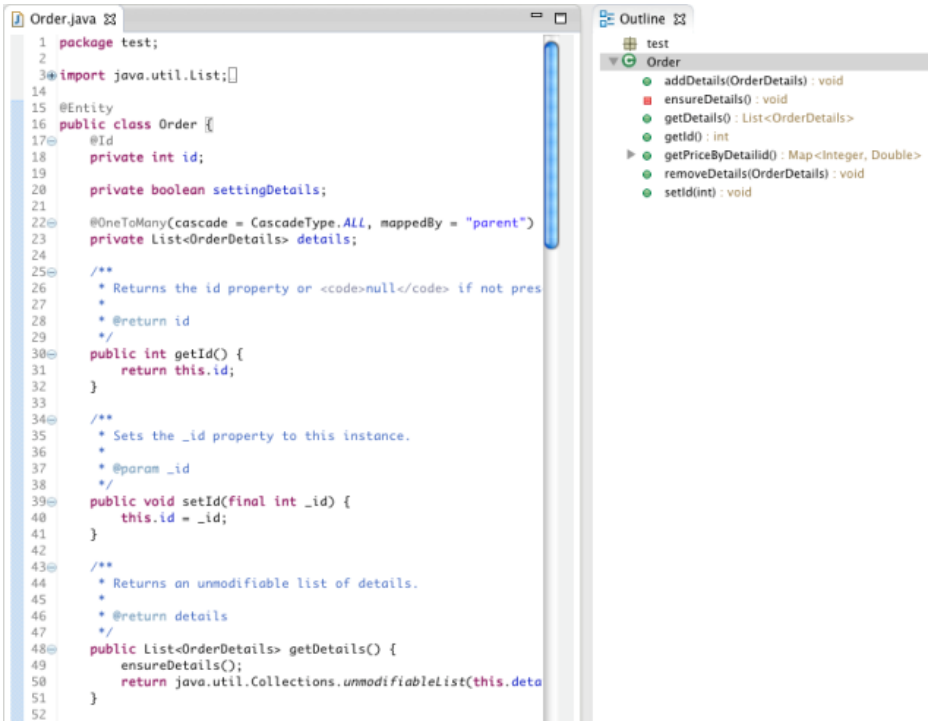

Figure 13. JPA entity generated with Entity-DSL

1.5.5. IntelliJ IDEA's Persistence Mapper

IntelliJ IDEA is an Integrated Development Environment some developers use to code in programming languages such as Java, Scala or Groovy. In the free version, the *Community* edition, database tools are not supported. However, in the paid version, the *Ultimate* edition, they are. In the official webpage they provide documentation on how to generate the persistence mapping by the database schema.

The user must create a data source with the database configurations, and once the persistence project has been created, he/she can import the database schema and ask the IDE to generate automatically a single XML mapping file, or an XML mapping file per entity.

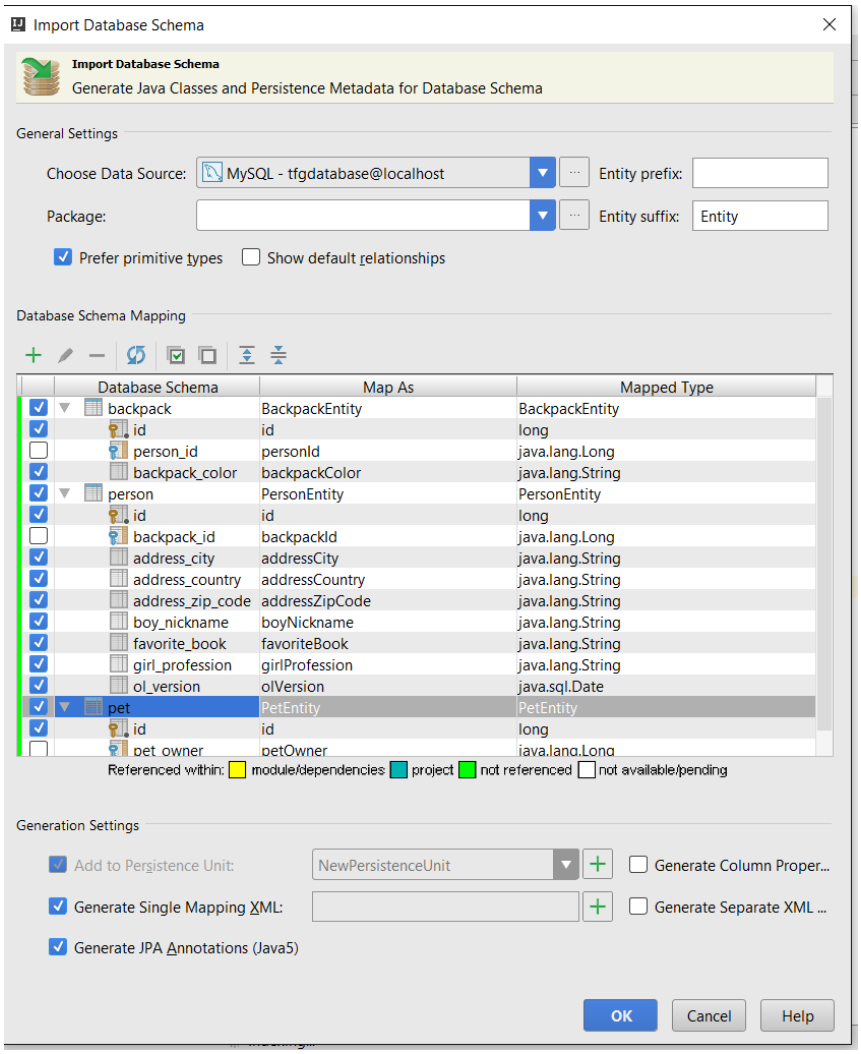

Figure 14. IntelliJ IDEA's import database schema feature

Once the process finishes, the persistence mapping XML file is created. Figure 15 shows a piece of the configuration file created as an example.

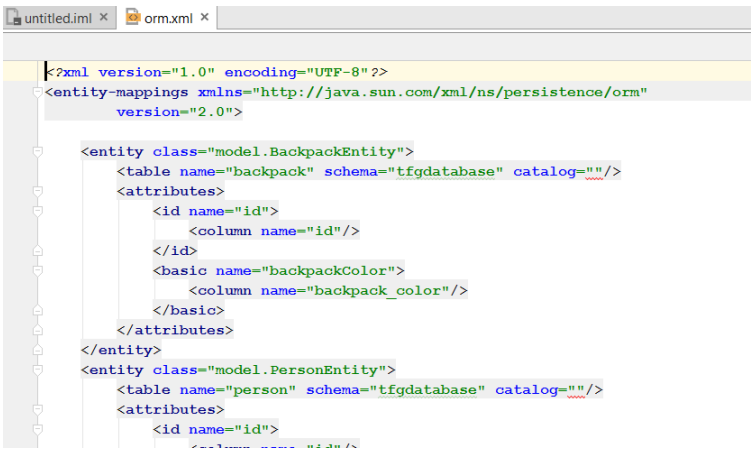

Figure 15. ORM.xml file generated by IntelliJ IDEA

1.5.6. Eclipse UML Generators

Eclipse, another *Integrated Software Development* or IDE, has created the Eclipse UML Generators projects that users can integrate with their Eclipse workbench.

Using these products, developers can create UML model files from their source code, and vice versa, they can generate automatically source code from UML model files.

There is substantial support and a large online community behind Eclipse that may be helpful for those developers to choose to use these projects. Even though they offer reverse engineering services that would allow users to create the UML model file from java classes, most users cannot seem able to find this feature or do not know how to use it.

The installation is free and it's done directly in the Eclipse IDE. After it's installed, the UML model file shows a new menu that offers the option to generate java classes.

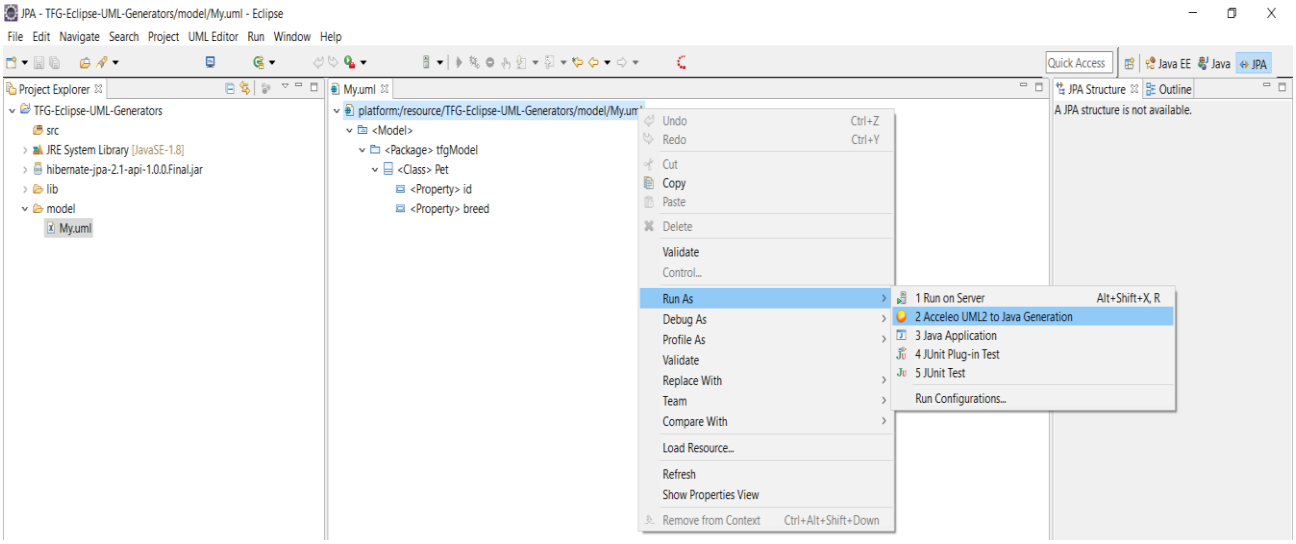

Figure 16. Eclipse UML generators feature

The tool will generate java entity classes with the information from the UML model file. If an error is raised or some mandatory information is missing, the tool does not inform the user and he/she will need to look for help and solutions online.

e

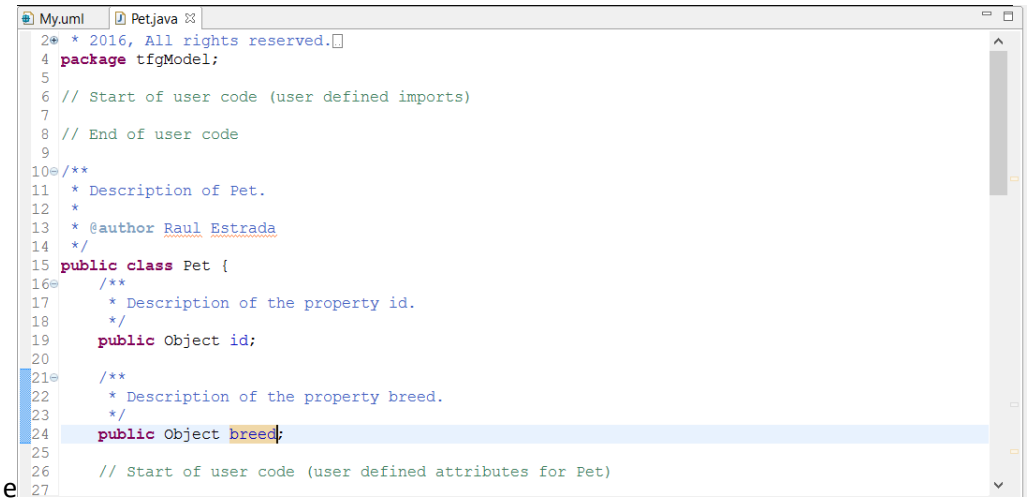

Figure 17. Java entity class automatically generated

The tool does not generate persistence annotations, nor does it automatically create the persistence ORM XML configuration file. It only creates the java classes.

1.5.7. Comparison of alternatives

Finally, a comparison of the different alternatives can be seen in Table 1, along with the main features and necessities identified by this project and provided by the different alternatives analyzed.

	SQL2JAVA	CodeTrigger	MetaEdit+	Entity-DSL	IntelliJ IDEA	Eclipse UML Generators	TFG Project
Load model from UML						✓	✓
Load model from database	✓	✓			✓		✓
Load model from source code							✓
Express model with DSL			✓	✓			✓
Generate ORM					✓		✓
Generate source code	✓	✓	✓	✓	✓	✓	✓
Generate UML model							✓
Generate database scripts							✓

Table 1. Comparison of alternatives

As the reader can notice from the previous comparison, there are some tools that focus on mostly one aspect of the problem: loading and analyzing the domain model from a given form, and generating source code automatically that represents such domain model. Moreover, these products usually become too complex to use or configure for the task in hand. Should the developer need to load the model and obtain different forms of it, he/she would need to install and configure several different tools, learn how to use them and execute them to obtain the different outputs needed.

To illustrate the previous point, imagine a development team where the domain experts express the essence of the domain model with a DSL. From that DSL, the development team will need to generate the source code representing the model (ideally with no persistence annotations), the persistence XML configuration file, the script to populate the database with the entities and relationships of the domain model, and maybe a UML model file for design purposes. With the different alternatives, they can use a tool to generate the source code, and maybe look for other tools to generate the other domain elements from there, or perform those tasks themselves, dedicating a substantial amount of time they could spend focusing on other and more important tasks.

Furthermore, some *Integrated Development Environments* offer the option of creating a persistence XML configuration file from the annotations found in the model or asking the user to introduce the information about the entities and properties, but they usually do not remove the annotations from the source code. Hence, the domain model still has persistence concerns in it.

.

2. Technical concepts

2.1. Domain-Driven Design (DDD)

Domain Driven Design, or DDD, is a software design approach where the design and implementation of the domain model is a shared responsibility between developers and domain experts. Three main concepts are related to this approach: Bounded Contexts, Single Responsibility Principle (SRP) and Ubiquitous Language.

First, a shared common language, called ***Ubiquitous Language***, is established between the developers and the domain experts. It's very important to drive design, implementations and discussions about the domain using such language. It is agreed upon by the team, no translations are needed and it represents a clear and understandable way of communicating the domain model. Every person in the team should use this language so that a common understanding is achieved and no information is lost in communications and translations.

The ***Single Responsibility Principle***, or SRP, related to the concept of high cohesion previously explained in the introduction of the project, means that software elements and domain model should be designed in a way where each of these units or elements should have only one responsibility or concern, or a set of highly connected responsibilities. Similarly, a software element (class, module, etc.) should therefore have only one reason to change. For instance, a module that handles how information is displayed to the user on the web browser and also performs database queries to retrieve personal information of users does not follow the Single Responsibility Principle. It has two different responsibilities: representation of information and data access. The module will need to change if a modification is needed for its representation concern, and it will also need to change if a modification is needed for its data access concern. By following the Single Responsibility Principle, future changes and maintainability are easily achieved and supported. The domain model should follow the SRP as well, and it should not include other concerns related to presentation, infrastructure or persistence.

Bounded Contexts are boundaries established around a given domain or sub-domain. It's a portion or partition of the domain. A domain makes sense inside of its corresponding bounded contexts, and inside that bounded context a Ubiquitous Language is established. The boundary around the model needs to be explicit and clearly defined. They are almost like different solutions (for example, in a large software system specialized in ecommerce there might be a Production Bounded Context, a Sales Bounded Context, a Shipping Bounded Context, etc.). An entity of concept in a given bounded context is not usually the same in another different bounded context. Each bounded context has its own language, domain model, rules and types.

In Domain-Driven Design, the domain model is a fundamental aspect of the product, it's the core of the application. Not only is the rest of the system built around it, but there is also a close connection between the domain and the source code expressing and representing it. Therefore, changes in the code will probably carry changes in the model, and vice versa. The domain model must be a strong foundation the application relies on and builds upon.

Some building blocks and key concepts in this approach used to model the domain are *entity*, *value object* and *custom types*. Entities are objects with an identity used to tell them apart and participate in relationships. Value objects do not have identities, they do not change and they are not part of relationships. They store values. Custom types are new personalized types used for attributes in either entities or value objects.

2.1.1. Domain and persistence model, and mappers

DDD requires developers to change their database-driven focus when designing and implementing the domain model. Developers tend to design domain entities thinking about how persistence will be implemented. This way, the domain model ends up being just a data placeholder for the persistence process. The domain becomes anemic, that is, it just carries or holds data retrieved from the database, rather than displaying the rich business behavior it should. Developers should focus on the business problem, and persistency is not part of the business problem. The design of the domain model should be done carefully since the output will carry repercussions throughout the rest of the application's lifecycle.

Therefore, the domain model and the persistence model should have two separate and different concerns. While the former models behavior and best represents the business domain, the latter models storage structure and deals with what information is going to be stored and where. DDD aims at creating a domain model rich in behavior that makes sense in a given context boundary and that uses the shared Ubiquitous language established for that domain model. This model shouldn't contain infrastructure elements and it should only depend on other possible domain models. If the persistence technology changes or the database is modified for any reason, like achieving better results, changes are only needed in the persistence model. The domain model, the core of the system, remains intact.

As a result, the impact of changes is considerably reduced and the domain model can be reused in other applications that could map data from this model into other different persistence models. It should be noticed that the persistence model depends on the domain model, and not the other way around. The domain model is independent. In DDD, in this situation the domain model is called upstream since it influences and impacts the downstream, the persistence domain.

Ideally, we could map directly from the domain model to the persistence model. Nonetheless, in practice this is usually not possible and some transformations are needed. Because of the different designs in the domain and the persistence models, and because the former one is behavior-driven and the latter one is storage-structure-driven, some translations or mappings must be. This is where mappers are helpful since they organize the information and do these translations.

Succinctly, nowadays in software development it's not unusual to have a very anemic domain model, and a very big business logic layer with lots of big services. This is because of developers' service oriented minds. What's more, persistence concerns are introduced in the model because of developers' database driven minds. The domain, the core and heart of the application, is compromised and not well designed. With DDD, the focus is placed first on the domain model, which will express and represent the business domain and business logic. Developers and domain experts will come together for this purpose. In the end, the development team will achieve a good and proper domain design rich in business behavior, software elements will be highly cohesive and the application business layer will be thinner, since most of the business logic is not in the domain.

Figure 18, taken from the blog post "Domain-Driven Architecture Diagram"[2], provides a diagram with an overview of the main concepts and ideas behind Domain-Driven Design.

[2] http://blog.ntcoding.com/2015/08/domain-driven-architecture-diagrams.html

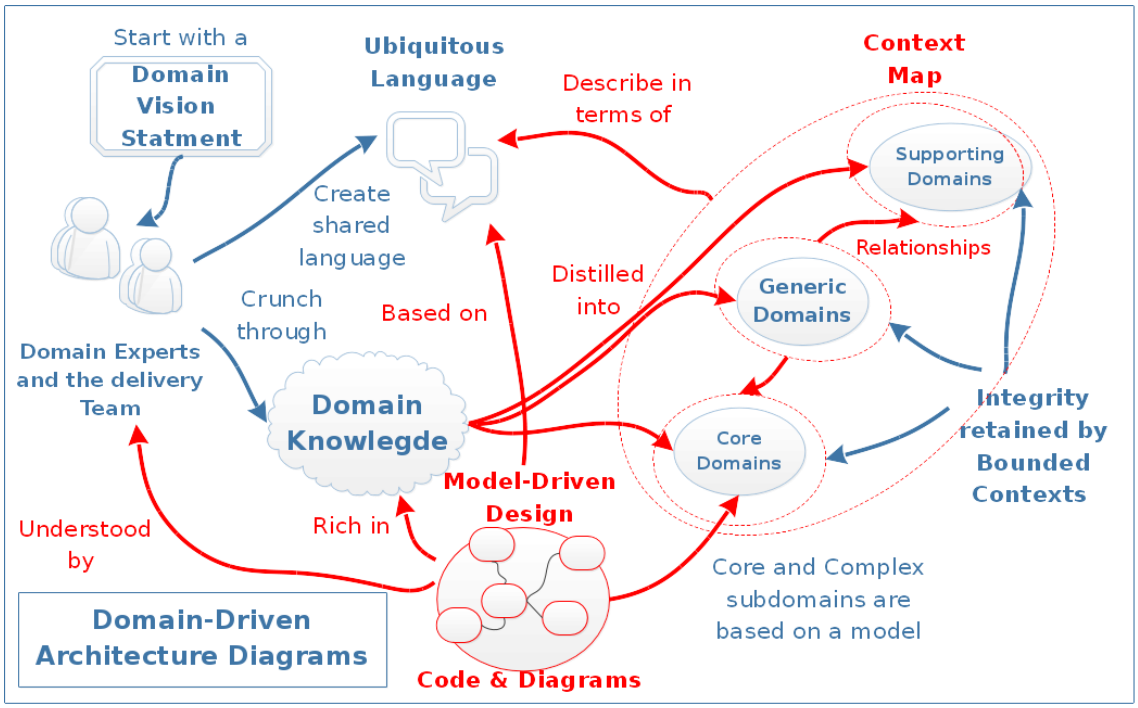

Figure 18. Domain-Driven Design diagram

2.2. Java Persistence API (JPA)

In the software development industry, the use of relational databases is widely spread because of their many benefits. They are a very mature technology, so the number of errors and bugs is very small compared to new databases that have been released as alternatives to relational databases. Moreover, they have been tested and used extensively for years, and many projects and companies already use them and changing and migrating to another type of database requires time and resources many companies are not willing to spend.

Relational databases also reduce data duplication, information is stored only once and they provide higher security levels and services. Furthermore, the database schema or database structure is the same for instances or records of the same table, so it's easier for developers to know what information a record contains, and what type its attributes are, because it's the same for every record of a given table.

The query language to insert and extract data from the database is also much more mature and stable. Relational databases use the SQL language, which is a standard in the industry, and the differences for different types of relational databases are minimal. The same cannot be said for data manipulation languages for other types of databases such as NoSQL or XML databases.

What's more, relational databases work well with transactions and attributes related to database transactions like Atomicity, Consistency, Isolation and Durability (these are usually called ACID) and they provide referential integrity, which means that information stored in different tables but connected through a relationship will be in a consistent state.

Some drawbacks of relational databases other alternatives are trying to fix or improve are related to performance and flexibility of the schema. The performance of relational databases is substantially damaged when data is spread across many tables and multiple join operations must be performed to query and extract data. Because the schema is the same for every record of a table, developers cannot add a new property to just a set of records. They either add the property to all the records and keep the column empty for many of them, increasing the table size, or they create a new table for the new type of object needed. This lack of flexibility is especially important in applications that require this kind of flexible database schema. With relational databases, a lot of tables will need to be created, and most of them will contain just a few records.

Once the reader understands why relational databases are so widely spread and used in the industry to this day, it's important to focus now on the code aspect of the application and the Object-Oriented programming paradigm. OO has gained a lot of popularity and it's safe to say it's one of the most used programming paradigms in software development.

With Object-Oriented programming, developers design domains and applications by breaking big problems into smaller and more manageable problems, identifying related objects or entities and encapsulating their data, state and behavior into objects that interact with each other. It fits better with how people abstract and design problems in the real world.

Objects and classes can be thought of as "units". They communicate with each other through public behavior they offer. This modularity increases code readability and maintainability. It makes developers go into a design phase before starting to code big scripts that will be more error prone and difficult to maintain and understand. In the end, the source code is cleaner, reusable, less error prone and easier to debug and more flexible and extensible. Big problems are divided into smaller problems and developers create modular solutions. Encapsulation is also a big part of Object-Oriented programming and developers must decide what data and behavior is exposed to the outside world, and what remains private.

Because of these benefits, it's extremely used and popular in software development. However, because the persistence takes place using a relational database and not an Object-Oriented database some transformations or mappings must be done. The objects created in the model do not map directly to relational databases, and the same holds true the other way around.

Object-Relational mappers are used to do these translations. They take the domain model objects in memory and transform them into the relational database form expected. And they take the data in the relational database and create the different domain model objects in memory representing that data. This way, the persistence model and the domain model remain independent. This process is transparent to the developer and it allows the development team to keep using and taking advantage of the Object-Oriented paradigm, and persist data in relational databases, whose benefits have been previously commented. Moreover, they have been optimized to increase performance, reduce the lines of code needed, they offer a data manipulation and querying language, and they can be used independently of the database.

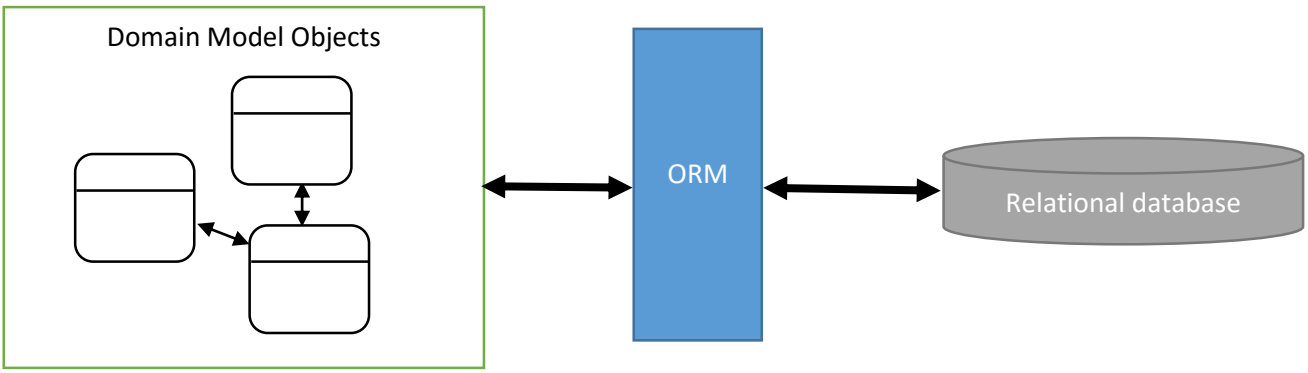

Figure 19. Basic diagram of ORM's functionality

In java, JPA (*Java Persistence* Interface) is a specification used to handle, manage and perform these mappings between objects and relational databases. With JPA, Object-Relational mappers must be configured properly to map the OO domain model to the entities and relationships in the relational database, and there are two common ways of doing so: with annotations or with XML configuration files.

Annotations are structures that provide metadata about an element. They are placed in the source code next to the element they are configuring or giving information about, such as classes, attributes, methods, variables or parameters. Annotations start with the @ symbol and they can contain other elements and annotations nested inside them that provides further information. In Figure 20 there is an example using annotations to configure some persistence aspects of the Employee class

```
@Entity
@Table(name = "t_employee")
public class Employee implements Serializable{
       ...
}
```

Figure 20. Basic example of annotations

Annotations were introduced in the Java programming language in 2004 and nowadays they are used for many purposes. In the previous example, the annotation *@Entity* indicates that the Employee objects are entities and therefore will be persisted as an object of the model. The *@Table* annotation contains the *name* attribute element with the name of the database table used to store the Employee entities.

XML files are formed by elements with attributes, written between with tags. The structure, elements, attributes, types and possible values of an XML file are defined and established in another XML file called XML schema. If the elements, attributes, the order they are placed or the values they contain do not match with the rules in the XML schema, the file will not be validated.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings version="2.1"
xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence/orm
http://xmlns.jcp.org/xml/ns/persistence/orm_2_1.xsd">
  <named-query name="Manager.findByIdentifier">
    <query>SELECT m FROM Manager m WHERE m.managerIdentifier= :empId</query>
  </named-query>
  <named-query name="Manager.findByDepartment">
    <query>SELECT m FROM Manager m WHERE m.department.id = :dptId</query>
  </named-query>
</entity-mappings>
```

Figure 21. Basic example of XML file

2.3. Domain-Specific Language (DSL)

Domain-Specific Languages, or DSLs, are languages designed and specialized to work with or in the context of a given domain. Opposite to Domain-Specific Languages Are General-Purpose Languages, which are computer languages designed to work with multiple domains. Examples of the latter are programming languages such as Java, JavaScript, C# or C++. Developers can use these to implement applications working across many different domains. Examples of Domain-Specific Languages are regular expressions or the database query language SQL. These languages are much smaller and they work inside a given precise and specialized domain.

Because DSLs are “personalized” and especially designed, they express ideas, concepts and semantics better. They are more powerful and descriptive inside their bounded context. However, they are barely useful in other contexts or domains. For this reason, they provide many benefits when used in a concrete part of an application, but typically they cannot be used for the whole system.

To illustrate this, let’s imagine a web system that manages travel agencies. The accounting module is quite complex and instead of working with a general purpose language, a new smaller language is created. Since DSLs are especially designed, they can describe a language readable and understandable by non-technical people. This way, the web system ends up with a powerful small language, understood and potentially designed by both domain experts and developers that better expresses the accounting part of the system and is rich in accounting domain knowledge. This language will probably understand concepts such as invoice, accounting seat or reimbursement. Validations and checks that make sense in that context can be built into the DSL. While this enriches the accounting module of the system, it’s not that useful outside this domain.

They are human readable, and when designed correctly domain experts can use these languages as well, and productivity in the development team should increase. However, they remain machine languages, and as such they must be executable by machines, and integrated with the rest of the application. Otherwise the module is semantically richer, but useless. Moreover, because they are created for specific purposes, they are easier to use and understand, and provide the bare minimum necessities. In the accounting DSL example, the

language will probably lack additional services offered by GPL like Java not related to accounting, like filesystem services, for instance.

There are two different types of DSLs:

- Internal: They are part of the host General-Purpose language. The code in the internal DSL is usually valid code in the host GPL, but it provides a small and specific number of features and properties that makes it seem like a different language.
- External: They are independent of the host General-Purpose language. They can be parsed independently and used by many different GPLs, and they have their own custom syntax.

Among the many benefits of DSLs, an important one is the integration of domain experts into the development team. The language is one they understand, can read and modify. As a consequence, a better communication is established, the quality of the domain is improved and these experts can even build domain validations in the language.

On the other hand, DSLs also have some important drawbacks to take into account. A new language must be defined, constructed and learned, increasing the budget of the project and adding additional time to the project schedule. Even though there's a lot to gain from these specific languages, the benefits are limited to the module the language will be used in. Moreover, because they are especially designed languages, no help will probably be available online, the language must be personally optimized and integrated with the host GPL and with the rest of the application.

2.3.1. DSL processing

Once a DSL file or script has been created using the new specific language, this file is parsed and a syntactic tree is created (a tree diagram with the input structure). From this syntactic tree, a semantic analysis is performed and a semantic model is created from the different elements and contents of such file and the syntactic tree. This semantic model will be created differently depending on the structure and grammar of the language. A DSL has a syntax and a semantic model, and the semantic model should support the language syntax. Using the language grammar, developers specify valid syntax.

Furthermore, additional elements such as generators can be built and added to the DSL so that when a DSL file is created or modified, these generators are executed and target output files are created. This way, developers can implement custom code generators that take the semantic model built when the DSL file is parsed, and generate source code files.

Figure 22 shows a small part of the grammar of the custom DSL designed and created for this project.

MyDsl.xtext

```
grammar org.xtext.example.tfg.MyDsl with org.eclipse.xtext.xbase.Xbase

generate myDsl "http://www.xtext.org/example/tfg/MyDsl"
import "http://www.eclipse.org/xtext/xbase/Xbase"

MODEL:
    elements+=ELEMENT*
;

ELEMENT:
    ENTITY | VALUEOBJECT | ENUMERATION | LINK
;

ENTITY: 'entity' (abstractEntity?='abstract')? name=ID ('extends' parent=[ENTITY])? '{'
        entityElements+=ENTITYELEMENT*
    '}';

ENTITYELEMENT: ANNOTATED_ATTRIBUTE | OPERATION;

VALUEOBJECT: 'value' name=ValidID '{'
        valueattributes+=ATTRIBUTE*
    '}';
```

Figure 22. Extract from DSL grammar

In the grammar file the valid syntax is described. Root elements can be ENTITY, VALUEOBJECT, ENUMERATION or LINK. A value object is defined with the keyword *value*, followed by the value object name, and a list of attributes surrounded by curly brackets.

Figure 23 shows an example of a DSL file created using the DSL of this project and the grammar of Figure 22. After parsing the file, the semantic model would resemble something similar to Figure 23.

sample.tfg

```
entity Student {
    @Id Long identification_code
    String full_name
    operation Integer calculateAverageGPA()
}

entity Course {
    @Id String code
    Integer number_of_credits
    String description
}

link Enroll {
    Many Student student
    Many Course course
    Integer mark
}
```

Figure 23. Example DSL file

Figure 24 shows the semantic model created from the DSL file in Figure 23.

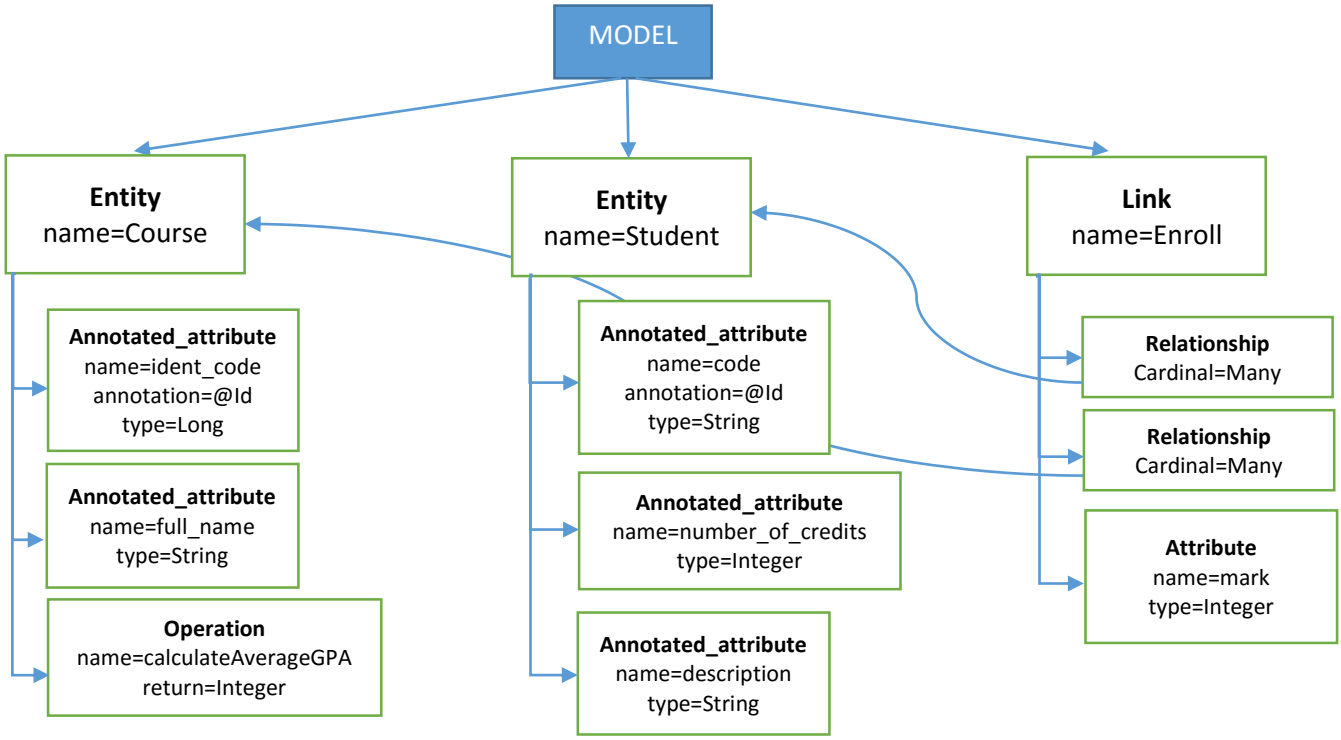

Figure 24. Example of DSL semantic model

2.4. UML

UML, Unified Modeling Language, is a language used to model and document software applications and systems through visual representations or diagrams that provide different views of the element being modeled.

It provides different diagrams that can be used to show the structure of a component or system, the behavior of the component or system and interactions. This way, different types of diagrams can be established: static and dynamic diagrams. Examples of static diagrams are class diagrams, package diagrams or component diagrams, among others. These diagrams have been introduced to the reader in the

Design section of the document, page 59. They represent structure and relationships. On the other hand, dynamic diagrams like the sequence diagrams display interactions, and use cases diagrams show behavior.

UML can be used to model elements and components that are not software related as well. Moreover, it is a general-purpose language.

Some important organizations in the software industry such as the Object Management Group and the International Organization for Standardization have adopted the UML language as standard.

In UML there are two different elements that are commonly thought of as the same: a model and a diagram. A UML model contains the different elements, their composition, types and multiplicity, and relationships, that form the domain model. However, a diagram is just a visual representation of a UML model. A UML model expresses the domain model and does not need a diagram, but a diagram is just a representation of the UML model, therefore a UML diagram needs a model.

Figure 25 shows a short UML model file generated automatically by the project. In this file, a root package called “com” contains several primitive data types, and another nested package called “restrada”, which contains a nested package called “domain”. The “domain” package contains a class called “Student” with its attributes “id”, “ol_version” and “universityIdentificator”. Furthermore, it also indicates that a UML profile has been applied and a reference to such file is provided.

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="20131001" xmlns:xmi="http://www.omg.org/spec/XMI/20131001"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
xmlns:ecore_1="http:///schemas/ecore/_rmIyAOODEeWUHdKrG-uHWg/2"
xmlns:uml="http://www.eclipse.org/uml2/5.0.0/UML"
xsi:schemaLocation="http:///schemas/ecore/_rmIyAOODEeWUHdKrG-uHWg/2 ../../TFG_FINAL/tfg-annotation-
processing/core/src/Ecore.profile.uml#_rmJZEOODEeWUHdKrG-uHWg">
 <uml:Package xmi:id="_Pn7ocOhqEeWmXaSKkS816w" name="com">
  <packagedElement xmi:type="uml:PrimitiveType" xmi:id="_Pn8PgOhqEeWmXaSKkS816w"
name="java.lang.Integer"/>
  <packagedElement xmi:type="uml:PrimitiveType" xmi:id="_Pn8PgehqEeWmXaSKkS816w"
name="java.lang.String"/>
  <packagedElement xmi:type="uml:PrimitiveType" xmi:id="_Pn8PguhqEeWmXaSKkS816w"
name="java.lang.Boolean"/>
  <packagedElement xmi:type="uml:PrimitiveType" xmi:id="_Pn8Pg-hqEeWmXaSKkS816w"
name="java.lang.Long"/>
  <packagedElement xmi:type="uml:PrimitiveType" xmi:id="_Pn8PhOhqEeWmXaSKkS816w"
name="java.util.Date"/>
  <packagedElement xmi:type="uml:PrimitiveType" xmi:id="_Pn8PhehqEeWmXaSKkS816w"
name="java.lang.Byte"/>
  <packagedElement xmi:type="uml:Package" xmi:id="_Pn8PhuhqEeWmXaSKkS816w" name="restrada">
   <packagedElement xmi:type="uml:Package" xmi:id="_Pn8Ph-hqEeWmXaSKkS816w" name="domain">
    <packagedElement xmi:type="uml:Class" xmi:id="_Pn8PiOhqEeWmXaSKkS816w" name="Student">
     <ownedAttribute xmi:id="_Pn8PiehqEeWmXaSKkS816w" name="id" type="_Pn8Pg-hqEeWmXaSKkS816w"
isID="true">
      <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_Pn8PiuhqEeWmXaSKkS816w" value="1"/>
      <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_Pn8Pi-hqEeWmXaSKkS816w" value="1"/>
     </ownedAttribute>
     <ownedAttribute xmi:id="_Pn8PjOhqEeWmXaSKkS816w" name="ol_version"
type="_Pn8PhOhqEeWmXaSKkS816w">
      <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_Pn8PjehqEeWmXaSKkS816w"/>
      <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_Pn8PjuhqEeWmXaSKkS816w" value="1"/>
```

```
        </ownedAttribute>
        <ownedAttribute xmi:id="_Pn8Pj-hqEeWmXaSKkS816w" name="universityIdentificator"
type="_Pn8PgehqEeWmXaSKkS816w">
         <lowerValue xmi:type="uml:LiteralInteger" xmi:id="_Pn8PkOhqEeWmXaSKkS816w"/>
         <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="_Pn8PkehqEeWmXaSKkS816w" value="1"/>
        </ownedAttribute>
      </packagedElement>
    </packagedElement>
  </packagedElement>
  <profileApplication xmi:id="_Pn8PkuhqEeWmXaSKkS816w">
    <eAnnotations xmi:id="_Pn8Pk-hqEeWmXaSKkS816w" source="http://www.eclipse.org/uml2/2.0.0/UML">
      <references xmi:type="ecore:EPackage" href="../../TFG_FINAL/tfg-annotation-
processing/core/src/Ecore.profile.uml#_rmJZEOODEeWUHdKrG-uHWg"/>
    </eAnnotations>
    <appliedProfile href="../../TFG_FINAL/tfg-annotation-
processing/core/src/Ecore.profile.uml#_WGFA8OMiEeWUHdKrG-uHWg"/>
  </profileApplication>
 </uml:Package>
 <ecore_1:version xmi:id="_Pn82kOhqEeWmXaSKkS816w" base_Property="_Pn8PjOhqEeWmXaSKkS816w"/>
</xmi:XMI>
```

Figure 25. Example of a UML model in the project

This example shows the benefits of a UML diagram. A UML model is not always easily readable, and having a visual representation can help understand its contents.

Figure 26 is a UML diagram representing the UML model shown in Figure 25.

In this project, UML has been used to read .uml model files and load the model from them, and to generate a UML model file from the intermediate domain model loaded.

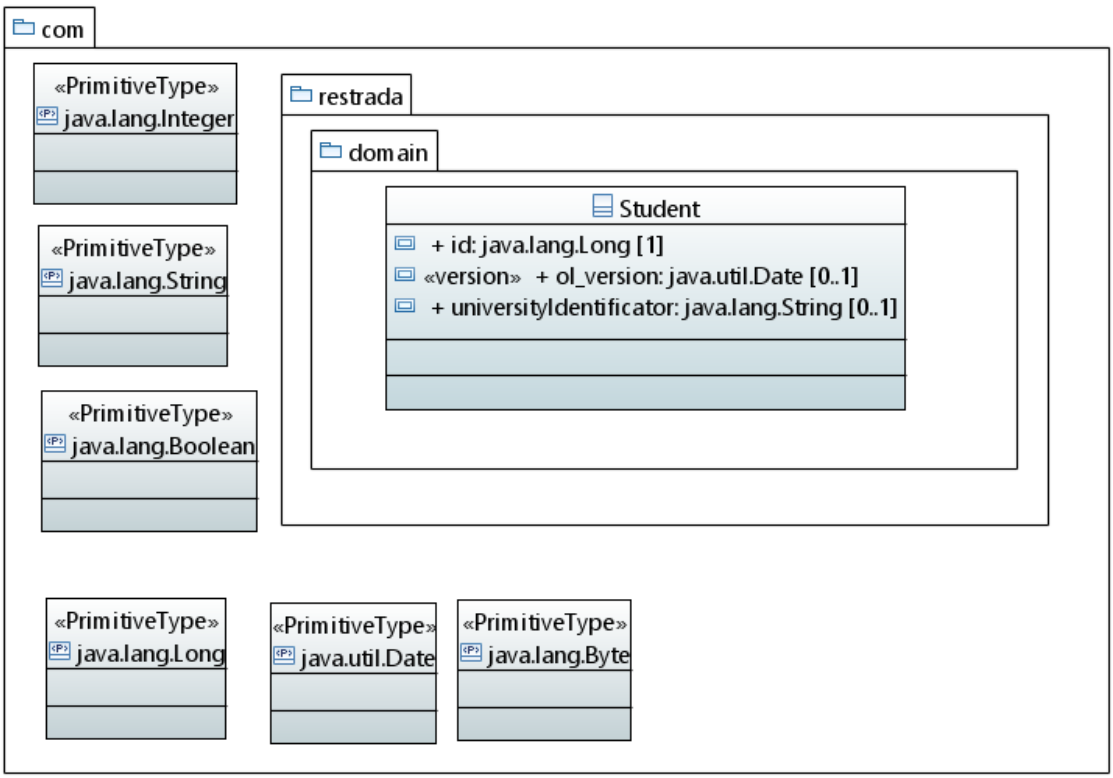

Figure 26. UML diagram for the UML model file

2.5. Transformations

This project performs several transformations from and to different forms or ways to express a domain model. Mainly, the following options are supported:

- **Load model from .class files**: Java .class files are files that have been compiled and contain bytecode, a language the Java Virtual Machine can understand and execute. The Java Virtual Machine is an abstract machine where java programs compiled into java files (.class) are executed. In Java, there's a component called ***ClassLoader*** that can take a given class and dynamically load it into this machine. Once loaded, different information or metadata about the class and its components are available to analyze. In this case, the important elements to look for are persistence configurations. Following this process, the domain tool processes the different .class files, looking for these persistence annotations, analyzing their contents and creating an internal model with the different domain elements (entities, value objects and custom types) found and the persistence configurations attached to them.
- **Load model from .java files**: .java files are source code files that have not been compiled. In this project .java files are important if they contain persistence annotations since that's the way to mark a plain java class as an entity, a value object, or a custom type, and specify persistence configurations for them and their elements. Using the compiler, however, the tool can transform these files into .class files and analyze them as briefly explained in the previous bullet point.
- **Load model from DSL**: DSLs were explained in Domain-Specific Language (DSL), page 50. Integrating the DSL into the main project tool, the tool can read the DSL file with the domain model, parse it and create the semantic tree (a diagram representing the essence and semantics of the domain model), and build an intermediate model with the elements and essence of that domain model.
- **Load model from UML model file**: UML (*Unified Modeling Language*) is a language whose purpose is to offer a standard way to model software elements or systems through different views, allowing to express structure or behavior. It's a language widely used in the industry and .uml files are files that contain information about domain model elements, their structure and relationships between them. These UML model files express the domain model, and are not very human readable. However, many software programs can be used to create visual representations of the domain model of an application by using the UML model file. The domain tool reads UML model files, analyzing and parsing their elements and information to create an intermediate model representing the domain model whose essence has been expressed in the .uml file.
- **Load the model from databases**: Databases are information repositories with a given structure or schema, and formed mainly by entities and relationships. In the database, metadata is also stored. Metadata is information or data about other data (like tables, columns, databases, procedures, users, etc.). This database metadata is especially useful when a user does not have information about the database schema but needs to get information about the database structure or composition. This is the case in this project. The domain tool accesses the database specified by the user. Once connected, it retrieves the database metadata and learns about the different tables in the database, their

composition, columns, table identity, relationships between tables and so on. As information reaches the domain tool, the intermediate model will be generated, filled with entities and relationships.

- **Generate database SQL scripts**: relational databases (those formed by entities and relationships in the form of tables connected) provide a data manipulation and query language called SQL. After loading the domain model and generating the intermediate model, the tool offers the possibility to create a SQL file with the code necessary to create the contents of the database (those entities and relationships). This file can be executed directly in the database.
- **Generate a UML model file**: UML model files were explained previously in this section. The tool offers the option to generate a UML model file that can be used to communicate the model to other systems, or to generate a graphical representation with a UML diagram of the domain model.
- **Generate the ORM file**: With JPA, persistence can be configured in two ways: annotations and an Object-Relational Mapping XML configuration file (ORM). As explained before, using the ORM file provides great benefits to the domain model, the code and the application in general. Once the original domain model has been loaded, the tool can automatically generate these ORM.xml file that may be used to configure persistence and object-relational mappings between the Object-Oriented model in the application and the persistence model in the relational database. Moreover, if the domain model was loaded from .java classes, new clean java classes will be generated from the original ones. That is, the source code is the same, but persistence annotations that were useful for developers but are not anymore and were jeopardizing the cohesion and quality of the model are removed and configuration is established with the newly created ORM file.
- **Generate clean java classes (skeleton)**: Once the domain model has been loaded, the tool can also generate java classes implementing such domain model. These classes will not contain persistence annotations. They are plain source code classes representing entities, value objects or custom types, their inner elements and methods to operate with them. Relationships handling is also provided. However, other business logic to be included and implemented in the domain is responsibility of the developers.

One product created for this project is a Maven plugin. **Maven** is a software tool that automates the build process of a software project. That is, it will execute different processes, download different elements and perform specific actions needed when compiling source code of the project, executing tests, cleaning the project, or building the project. When using a Maven plugin in a specific project, developers can specify that they want the plugin to be run when a given action or goal is executed. For instance, they could configure the domain Maven plugin to load the domain model from the java classes and create the ORM.xml and the database script file when the client project is compiled.

3. Design

3.1. Platform selected

Since this is an experimental project, several technological tools and platforms have been considered before committing to one. And even after committing to one tool or platform, in some occasions reconsiderations had to be made to see if switching to another technology or using another approach would be beneficial.

3.1.1. Platform and programming language

In its origins, this project was a JPA annotation processing tool. It would process persistence annotations found in java source code and then it would generate the ORM persistence configuration file. When the project was at that stage, the first decision was made: what platform and programming language to use (C#, Python, Java, etc.).

Java was the obvious decision. The project's purpose was to work with java source code files to scan and process Java persistence annotations. Moreover, while other languages such as C would provide benefits like better performance and more control over machine specific features, Java is multi-platform, or rather, platform independent. The only requirement would be to have java installed, but since the project would work with java source code files, java would need to be already installed in the end users' machines. Furthermore, Java is a mature and tested programming language already familiar to the development team, implementing the Object-Oriented programming paradigm and easy to learn. While these last reasons are not the main reasons considered when making the decision, they are important since the maintenance and extension of the products may be carried out by teams and developers other than the original developer that implemented the project.

3.1.2. Type of application: web application vs standalone desktop program

The next aspect to analyze and decide on was the type of application to create, and the main options considered were either developing a web application so that developers could upload their code and download the ORM.xml persistence configuration file, or implement a standalone desktop application. A standalone desktop application is a program that runs in the machine of the end user, instead of running on a server and be available through the web.

Web applications are very popular nowadays, the user wouldn't need to install or download any program, and it would be available to a much wider audience with less effort. Moreover, interoperability is achieved easily using web applications. However, web applications would offer additional features that are not really needed or required of this project, but it would increase substantially its complexity, budget and schedule. Servers would need to be purchased and configured. The business logic would need to be located in those servers, and all the clients of the web system would need Internet connection to access the different services offered. This does not make much sense in this project. A **standalone desktop application** has been chosen because it avoids these drawbacks while meeting the requirements and needs that justified the development of the project. Since no Internet is required, the application is available at all times, and performance is substantially improved

(no communication is send over a network). Furthermore, not that this application would be the target of any external cyberattack, but desktop applications are more secure and they avoid threats inherent to web applications such as denial of services.

3.1.3. Annotation processing approach

Computers understand instructions expressed using machine code, but humans do not understand easily this code, thus it would be quite difficult to program applications using these languages. Therefore, humans write applications using high level programming languages they understand and an intermediate processor (compiler) performs the translation from the latter to the former. The Java Compiler is a program that takes the source code files written by programmers using the java high level programming language, and translate them into Java class files with bytecode (the instruction set of the Java Virtual Machine). Because the Java Virtual Machine is a machine abstraction used by concrete computers to execute Java programs no matter the Operating System, and this machine understands bytecode instead of the Java programming language, it is necessary to use the compiler to translate the java files coded by humans into files with content the JVM can understand and run.

While the Java version J2SE 5.0, also called Tiger and released in 2004, provided a separate tool to process the newly added annotations feature, the next version, Java SE6 or Mustang, released in 2006, integrated the annotation processing feature in the java compiler. This improvement allows developers to add their own annotation processors to the java compiler, and by simply compiling the source code the annotation processors will be executed with no additional and intermediate tools.

Taking the previous information into account, another technological decision to be made was how to approach analyzing and processing the persistence annotations in the java source code model. Two options were considered:

- Create a custom annotation processor and attach it to the java compiler.
- Utilize a software element used to load compiled java classes into the JVM (*Java Virtual Machine*).

The former option would consist on implementing a custom annotation processor, configure the java compiler and when doing so, attach this custom processor so that, as the compiler is compiling java source code files, the processor will be executed as well. The Java Compiler (*javac*) found in the machine system is used to compile all the java files found in that location and all its subdirectories (subfolders). The custom processor is called and the annotation processing starts. However, this approach is more platform dependent than the alternative and it depends on a feature subjected to change.

The *ClassLoader* is a software element that takes a compiled java class (.class file) and loads it into the JVM so that it can be executed. The second approach exploits this element. It takes the compiled domain classes, loads them and looks for persistence annotations. It does not depend on a compiler feature. Moreover, before a decision was made new additional functionality was added to the project, including the possibility to load domain models from compiled java classes. If the custom processor was to be use to load the domain model from java source code files, another different module would need to be implemented to load the model from compiled classes, and no code could be reused. This is because the annotation processor would be executed when compiling classes, but already compiled classes would not be able to use this service.

However, if the *ClassLoader* is used, loading the domain model from compiled classes is as previously explained, and loading the model from java source code files is straightforward. The only thing needed is to add a previous step where the program would simply use the java compiler to compile the files, and then use the same loading module developed. Thus, development time and budget decreases, and code reusability and maintainability increases. Succinctly, the second approach, using the ***ClassLoader*** was used in the project.

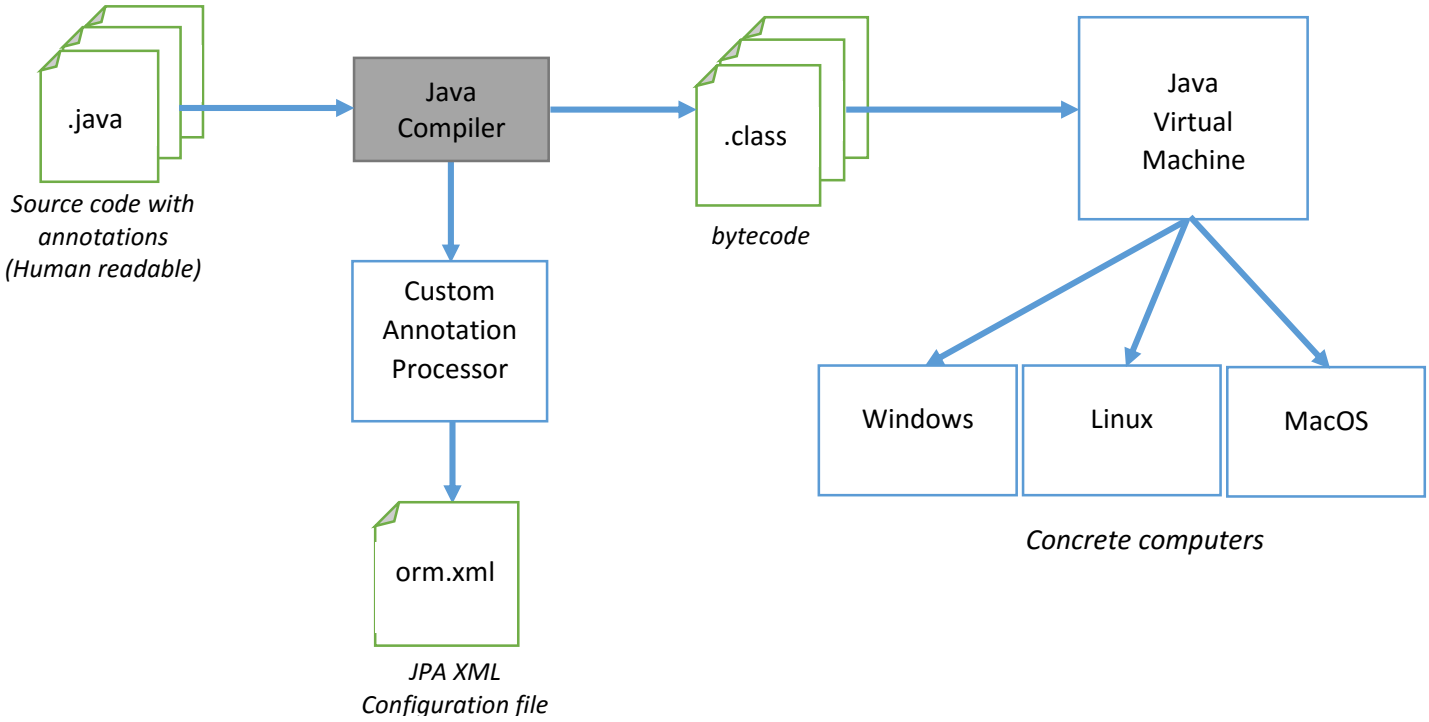

Figure 27. Overview of the compiler custom processor approach

3.1.4. Generating the ORM file: JAXB vs MOXy vs creating programmatically the XML file

The ORM persistence configuration file is an XML file, and this project constructs such file from the intermediate domain, loaded from the original domain model, and express through plain java classes.

Because the expected output is a file with the XML format, two alternatives were considered:

- Generate the XML file programmatically: construct a new software module or element that would traverse the different items in the intermediate model, creating and filling the XML file expected.
- Use an **XML binding approach**: Utilize an XML binding architecture or tool so that a mapping from the classes in the intermediate domain of the project to elements in the XML file is configured, and the transformations from the model to the XML file and vice versa are done automatically.

The XML binding approach was used eventually because it's an approach widely used in the industry, it saves development time and the new software element that would create the XML file in the other approach would not be as mature and tested as the XML binding tools available. Moreover, the number of lines of code required using the latter approach is much smaller and this method is less error-prone.

Once the XML binding technique was chosen, two specific tools were analyzed:

- JAXB reference implementation:
- MOXy, an Eclipse implementation of the JAXB specification.

JAXB, or Java Architecture for XML Binding, allows developers to annotate Java elements, like classes and attributes. This way, when a java instance of the annotated class is transformed into a format that is more suitable for transmission or persistence (*marshalling*), the object is transformed into an XML element. Inversely, *unmarshalling* refers to the process where the data suitable for persistence or transmission is transformed into the Java objects with the appropriate state. Once the Java classes are annotated to work with JAXB, the Java-XML binding works in both directions: when we marshal objects the XML document is created, and inversely, when we read the XML document (*unmarshalling*), the Java objects are created.

Examples of JAXB annotations can be seen in Figure 28. The @XmlAccessorType annotation controls how the serialization of the class elements will be done: by fields or properties. In this case it uses field access type, so by default every attribute that does not contain the static keyword and that is not annotated with @XmlTransient will be marshalled. Moreover, @XmlAttribute indicates that the attribute will become a XML attribute of the class XML element, and not a XML subelement of the class XML element.

The following example shows a class with an attribute annotated with @Version.

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "version", propOrder = {
    "column",
    "temporal"
})
public class Version {

    protected Column column;
    @XmlSchemaType(name = "token")
    protected TemporalType temporal;
    @XmlAttribute(name = "name", required = true)
    protected String name;
    @XmlAttribute(name = "access")
    protected AccessType access;

        …
}
```

Figure 28. Example of JAXB annotations

After marshalling, the XML document is created and the Version element is illustrated in Figure 29.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<entity-mappings xmlns="http://xmlns.jcp.org/xml/ns/persistence/orm">
  …
  <entity name="Student" class="com.restrada.domain.Student">
      …
    <attributes>
      …
      <version name="ol_version">
        <temporal>DATE</temporal>
      </version>
      …
    </attributes>
  </entity>
  …
</entity-mappings>
```

Figure 29. orm.xml example showing how the Version element is transformed using JAXB

In the end, the **JAXB reference implementation** was used because even though the Eclipse implementation MOXy offers additional features on top of the JAXB specification, it adds yet another dependency to another platform and development team behind this implementation. Moreover, using MOXy is not straightforward, which using JAXB reference implementation is. Furthermore, the JAXB reference implementation is a standard, common and agreed by the industry, and with a much larger community behind it.

3.1.5. DSL technologies and approaches

Different alternatives and approaches were considered as well to design and incorporate a DSL into the project.

- Implementing a custom compiler to read and parse the DSL grammar designed for this project.
- Create a DSL project with an already existing technology.

While the first approach allows the creation of a much powerful and flexible language and tool to work with it, it also requires much more development time and an increase in the budget of the project. This is unfeasible because the DSL needed for the project is and should be very simple and basic so that domain experts can work with it without any difficulty. Moreover, the only services needed from the DSL is the editor and the ability to compile a DSL file and get the semantic model. After analyzing technologies that support the creation of DSL projects, the latter approach was used even though it adds another technology dependency to the project.

Eclipse Xtext is an open source project offered by the Eclipse foundation that can be used to create a custom DSL easily using the Eclipse workbench. With a lot of information and tutorials, and a large community behind it, this tool creates DSL projects that incorporate features such as a DSL editor for the Eclipse workbench, and even platform-independent DSL editors. Moreover, the language can also be customized to contain custom validations and autocomplete options. Finally, XText allows the new DSL language to be mapped to java elements, and even use java elements such as types, blocks and control flow structures.

Succinctly, **Eclipse XText** was used to create the custom DSL for this project.

3.1.6. Xtend

Xtend is another general-purpose programming language, with origins in the Java programming language, that builds upon it to improve and offer additional services such as type inference, extension methods. It does not have statements since everything is an expression, and it compiles and translates automatically into java code.

Like Java, Xtend files are compiled, loaded and executed in the Java Virtual Machine, and it's a cross-platform language. It was created in 2011 and it's more human readable than java, since it provides a more concise syntax. This language was created with XText, a framework and software tool used to create and define custom languages and DSLs (Domain Specific Languages).

In this project both XText and Xtend were used to define and create the custom Domain-Specific Language for the project.

Figure 30 shows a piece of Xtend code that infers the java element from the DSL *VALUEOBJECT* element. In the example, the function creates a class with the name specified in the DSL file for the *VALUEOBJECT* element, and then it traverses its inner *ATTRIBUTE* elements to create java attribute fields and java methods to get and set the different fields.

```
92
93    def dispatch void infer(VALUEOBJECT element, IJvmDeclaredTypeAcceptor acceptor, boolean isPreIndexingPhase) {
94        acceptor.accept(element.toClass(element.name)) [
95            for (ATTRIBUTE attrib : element.valueattributes) {
96                members += attrib.toField(attrib.name, attrib.type)
97            }
98            for (ATTRIBUTE attrib : element.valueattributes) {
99                members += attrib.toGetter(attrib.name, attrib.type)
100               members += attrib.toSetter(attrib.name, attrib.type)
101           }
102       ]
103   }
```

Figure 30. Example of Xtend code used in the project

3.1.7. Why plugins for Eclipse and Maven?

One question that may need to be answered is why develop plugins for Eclipse and Maven. Why create a plugin for Eclipse and not for other IDEs such as NetBeans or IntelliJ, and why create a plugin for Maven and not for other build automation tools like Ant or Graddle?

The main goal and purpose of this project is to help developers and ease their work when dealing with domain models. Eclipse is one of the most popular, if not the most popular, IDE used. And Maven is one of the most popular, if not the most popular, build automation tool used. Their popularity varies depending on the source consulted, but many agree that they are on the top and it's indisputable that they are widely used.

Figure 31, an image taken from the website CodeAnywhere[3], shows the most popular IDEs and code editors in 2014. The first and second most popular are code editors, which are

[3] https://blog.codeanywhere.com/most-popular-ides-code-editors/

not relevant in our case. Regarding IDEs, the most popular is Eclipse, followed by NetBeans and IntelliJ.

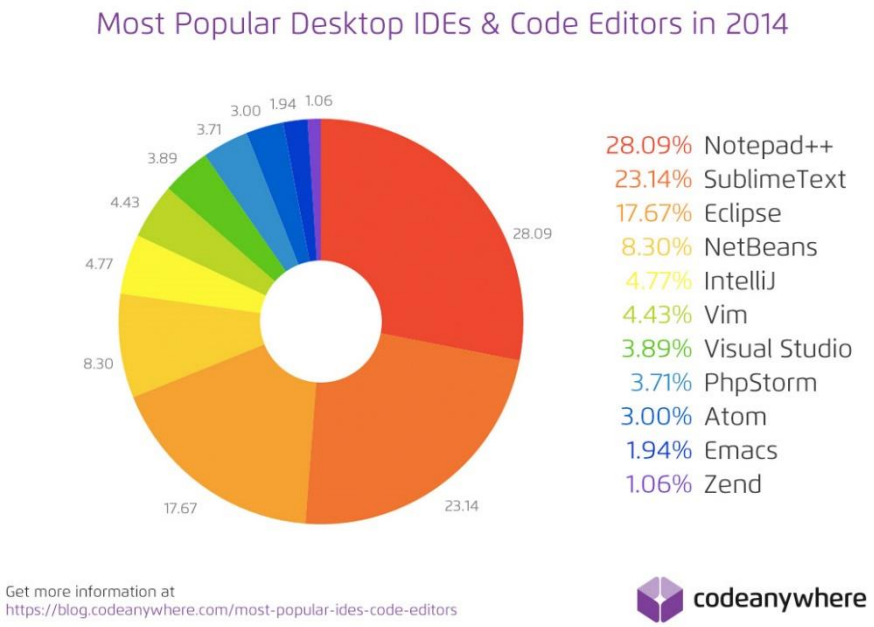

Figure 31. Most popular IDEs and code editors in 2014

Some websites, like ZeroTurnAround[4] or the blog CodeImpossible[5], have asked their readers which build automation tool they use the most, and the results seem to indicate that Maven is the leading tool.

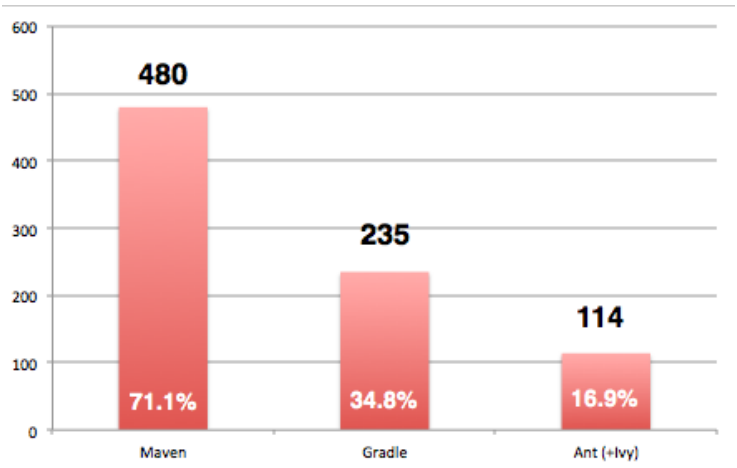

Figure 32. CodeImpossible survey results in 2013

[4] http://zeroturnaround.com/rebellabs/the-build-tool-report-turnaround-times-using-ant-maven-eclipse-intellij-and-netbeans/
[5] http://arhipov.blogspot.com.es/2013/08/java-build-tools-survey-results.html

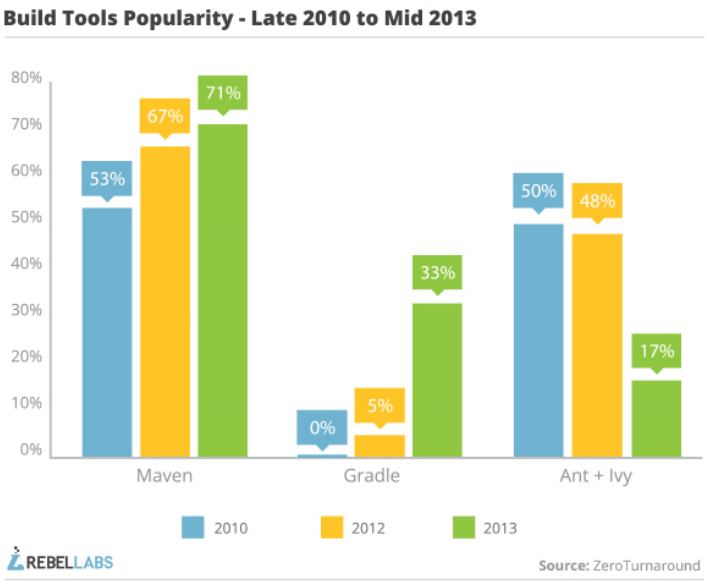

Figure 33. Build tool popularity according to the web ZeroTurnAround

3.2. System architecture

The software architectural solution is composed of a system divided in different "layers" or tiers. A software system using a multi-tier architecture separates the three main concerns and creates three different systems or modules: one for the presentation responsibility, one for the business logic, and the last one for data access. These software layers are built on top of each other in a way where the upper layers need and use the lower layers. To better understand the concept of layers, they could be thought of as layers in a cake, or onion layers. The presentation layer is the outermost layer and relies on the business layer, which is the one in the middle. This one relies on the data access layer, which is the innermost layer. With tiers, these layers do not need to be on the same application or the same system.

Since this project does not deal with data persistency, the architecture has two main tiers.

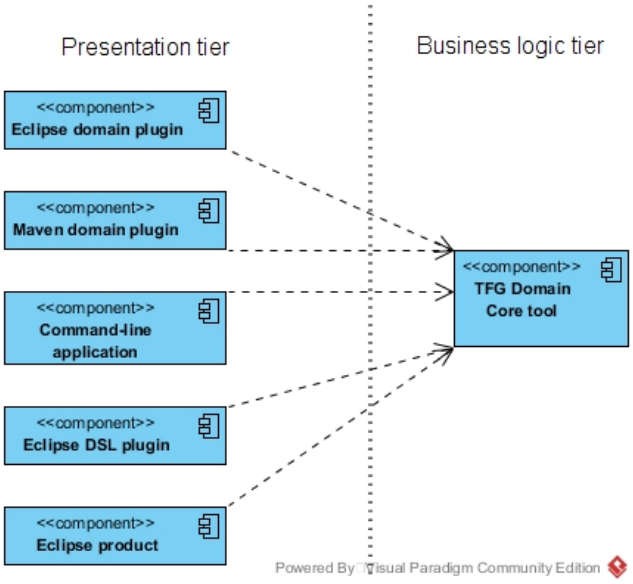

Figure 34. System tier architecture

These two tiers are:

- Presentation tier: This tier deals with user interaction and how to represent data to the user. Several products of the project fall under this tier: Both the Eclipse plugins, the Maven plugin, the command-line program and the Eclipse product. These products do not implement or execute any business logic, they are products that offer the different services to the user through different interfaces and environments they are integrated into, but they rely on the TFG domain core product to execute the user requests.
- Business tier: This tier implements and executes business logic of the system. In the project, the business tier is formed by the TFG domain core product. This system or product is the one that contains the services that load and generate models, and the DSL services.

As previously commented, there is no data access tier. Moreover, the presentation layer depends on and relies on the business tier to perform its tasks, but this dependency is unidirectional. The business tier does not and should not depend on outer layers or tiers such as the presentation one. Furthermore, each tier has only one concern or responsibility, and dependencies between layers is decreased by using an external interface so that the "client" component only knows the behavior externalized explicitly by the "provider" component.

This architecture increments encapsulation and reusability. By having a module in charge of business logic and other different modules in charge of presentation, the system is able to provide many different presentations to the user by just creating and adding a new presentation module, without needing to duplicate any business logic. Succinctly, maintenance is also improved because any change or modification needed is done only once in one single component.

3.2.1. Software patterns

Many general recurring problems arise when designing and implementing software and its architecture. As software architectures and engineers increase their knowledge and deal with solutions for these problems, they have put together generic solutions for them. Therefore, software patterns are general recipes built based on wide experience coming from a large number of professionals that have tackle these problems. Because they are general solutions, they must always be adapted to the different software applications.

The main patterns used in this project are the following.

3.2.1.1. Facade

In the *Facade* design pattern, a module identifies and exposes explicitly the behavior and services outer elements of the module can use and execute. The inner elements in the module with the facade may have different methods or services, and maybe also different implementations for the different services offered to the outside world, and they may call each other's public methods, but exterior software elements do not know anything about it. The module creates a façade the outside world sees, and its inner elements hide behind that facade.

Figure 35, an image taken from Five's Weblog[6], shows a graphical representation of this design pattern.

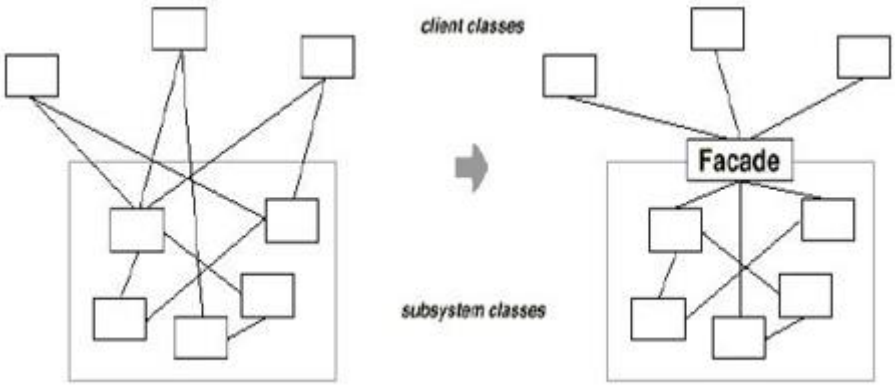

Figure 35. Facade design pattern diagram

An example of the Facade pattern being used is with the generation services offered by the TFG core domain tool. A java element called *interface* is used to specify a contract between providers implementing such interface, and clients of the interface. The ***Writer*** interface defines the following behavior to be externalized.

[6] https://powerdream5.wordpress.com/tag/facade-pattern/

```java
package output;

import java.io.IOException;

import javax.xml.bind.JAXBException;

import com.sun.codemodel.JClassAlreadyExistsException;

import model.Package;

public interface Writer {
    public void write(Package rootPackage, String outputDirectory) throws
    JAXBException, IOException, JClassAlreadyExistsException, ClassNotFoundException;
}
```

Figure 36. Writer interface

Elements outside the module have access to the Writer interface, and they can only use the write service offered, as shown in Figure 36. However, different classes implement this service in different ways inside the module, as shown in Figure 77.

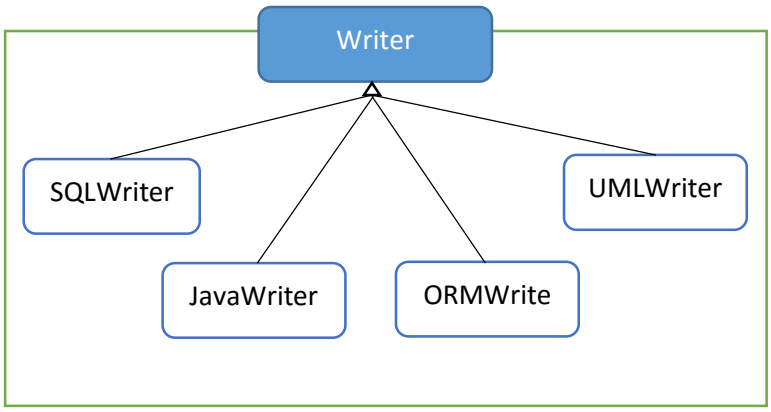

Figure 37. Facade example in the project

3.2.1.2. Factory

Sometimes a software element may want to create another software element, and the latter may have different java types because it's part of an inheritance relationship or because it uses interfaces. This can happen when the object expected is the parent class or the interface. The actual object received, however, may be the parent class, any of its children, or in the case of the interface, any class implementing the interface. The difference between receiving one class or another, even though the interface/parent class is the same, is very important since the child class will probably implement and execute the same service in different ways, thus obtaining very different results or carrying out the same service or business rule very differently.

The solution may seem straightforward at first. Expect the class you want to receive. However, this will reduce flexibility and may not work in many occasions. For example, when depending on some other variable the actual class must be different, but the service and behavior to be used is the same no matter the actual type.

With the Factory design pattern, a special class called *Factory* is created. This factory will offer different methods to create different elements, but returning the interface they implement. This way, creation of different objects but returning the same interface can be achieved without having to know anything about the implementation and without having to specify the type of the child class.

Figure 38, an image taken from the website Dofactory[7], shows an overview of the Factory design pattern and the elements involved in it.

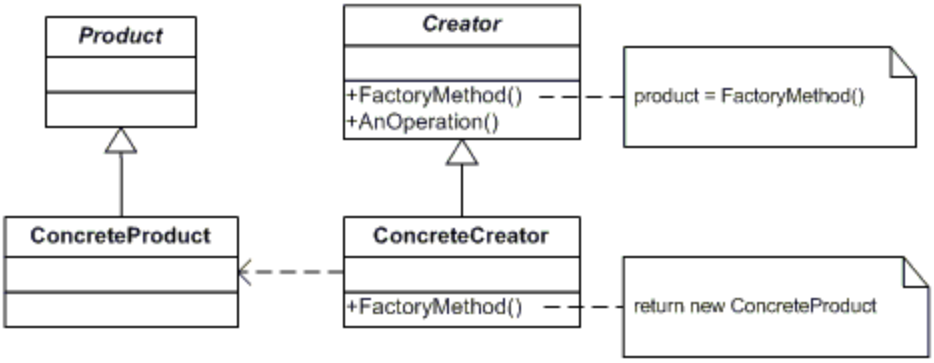

Figure 38. Factory design pattern diagram

An example of the Factory pattern in the project is shown next and will use the Writer interface and the classes implementing it, as the reader is already familiar with them from Figure 36.

A WriterFactory interface has been created and identifies the different operations that can be used to create a Writer element, but the actual java class obtained can be different. The WriterFactory interface is shown in Figure 39.

WriterFactory.java | WriterFactoryImpl.java

```
1 package output;
2
3 public interface WriterFactory {
4     public Writer getORMWriter();
5     public Writer getUMLWriter();
6     public Writer getOracleSQLWriter();
7     public Writer getMySQLSQLWriter();
8     public Writer getPostgreSQLWriter();
9     public Writer getHSQLDBSQLWriter();
10    public Writer getJavaWriter();
11 }
```

Figure 39. WriterFactory interface

An actual Java class named WriterFactoryIml implements the interface and its services, creating instances of the concrete classes implementing the Writer interface. This class is shown in Figure 40.

[7] http://www.dofactory.com/net/factory-method-design-pattern

```java
*WriterFactoryImpl.java
public class WriterFactoryImpl implements WriterFactory {
    @Override
    public Writer getORMWriter() {
        return new ORMWriter();
    }

    @Override
    public Writer getUMLWriter() {
        return new UMLWriter();
    }

    @Override
    public Writer getOracleSQLWriter() {
        return new SQLWriter(DatabaseType.ORACLE);
    }

    @Override
    public Writer getMySQLSQLWriter() {
        return new SQLWriter(DatabaseType.MYSQL);
    }

    @Override
    public Writer getPostgreSQLWriter() {
        return new SQLWriter(DatabaseType.POSTGRESQL);
    }

    @Override
    public Writer getHSQLDBSQLWriter() {
        return new SQLWriter(DatabaseType.HSQLDB);
    }

    @Override
    public Writer getJavaWriter() {
        return new JavaWriter();
    }
}
```

Figure 40. WriterFactoryImpl class. The concrete factory

A graphical representation of the elements of this project involved in this example of the Factory designed pattern is provided in Figure 41.

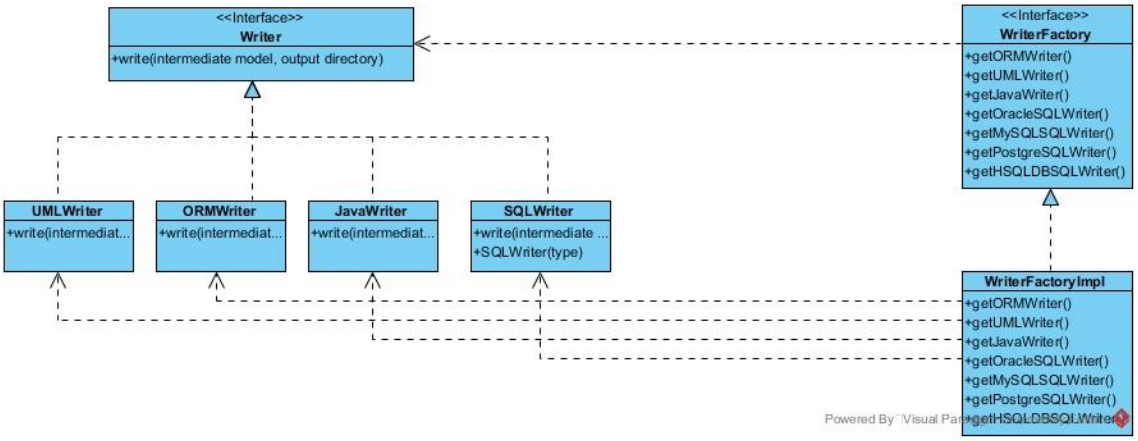

Figure 41. Factory diagram in the project

By calling the different methods of the WriterFactory interface on an instance of the WriterFactoryImpl class implementing it, the Writer expected can be a UMLWriter, a SQLWriter, a JavaWriter or an ORMWriter, and only the factory knows about the specific classes implementing the interface. Other classes using the factory element only know about the behavior expressed in the Writer interface (the *write* method).

3.2.1.3. Singleton

Sometimes some software elements should have only one single instance created in the application or system. As a consequence, there exists just that one access point to the

software element. If such instance or object is to be used by multiple clients, those clients will access and communicate to the same instance.

When a software element implements the Singleton pattern, no one can create an instance of the element. Clients will ask the software element statically and globally for that single instance, and the software element will see if it has been created previously. If it has, it just returns that instance, and if it has not, then that same software element may create an instance of itself.

Since this pattern is quite simple, no general diagram of the pattern is provided. Instead, an example of how the pattern is used in the project is shown next. The WriterFactoryImpl explained in the example in Factory, page 71, implements the Singleton pattern. It just does not make any sense to create multiple instances of the factory because only one is necessary. The different clients can use that one instance to ask for the different "writers" they need.

WriterFactoryImpl.java

```
9 public class WriterFactoryImpl implements WriterFactory {
10
11     private static WriterFactoryImpl INSTANCE = null;
12
13     private WriterFactoryImpl(){}
14
15     public static WriterFactoryImpl getInstance() {
16         if (INSTANCE == null) {
17             INSTANCE = new WriterFactoryImpl();
18         }
19         return INSTANCE;
20     }
21
```

Figure 42. Singleton in the Project

As the reader can see in Figure 42, a global instance of WriterFactoryImpl is established, and since the constructor ("method" used to create and instantiate an object of the class) is private, only that same class can call it. Clients can ask for the instance with the static *getInstance* method. A static method is one that is not called on an object of the class, but on the class itself (i.e. *WriterFactoryImpl.getInstance()*). If the instance has not been created previously, it is now. And the method returns that instance.

Therefore, the instance class and the inner class elements needed are shown in Figure 43.

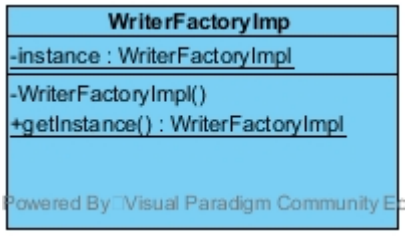

Figure 43. Class diagram of Singleton in the project

3.2.2. Deployment diagram

The following deployment diagram shows the system and subsystems of this project, and the hardware or devices where they will be installed and executed. Since the project

contains several products and they are integrated in different environments, different deployment diagrams are provided for them. However, all of them illustrate the same general concept: the product is installed along with the TFG Domain core product in the end user's device. In that device, they may communicate with other systems/applications.

3.2.2.1. Command-line product deployment diagram

Figure 44 shows the deployment diagram of the command-line application in the computer of the end user. The TFG Domain core product will be installed as well, and it will use the java compiler installed in the computer.

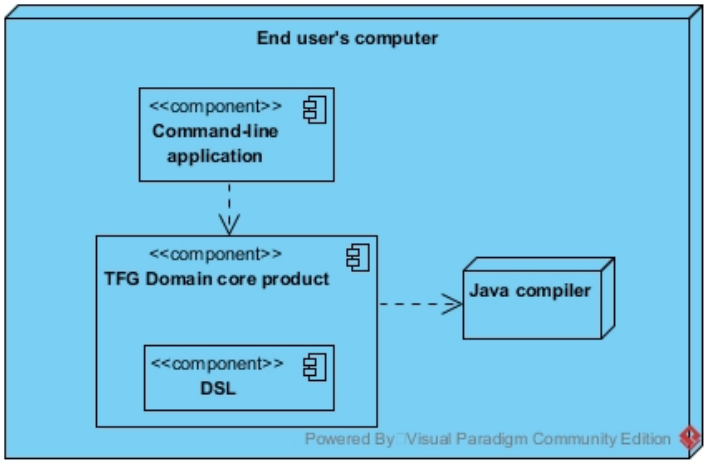

Figure 44. Deployment diagram of the command-line application

3.2.2.2. Eclipse domain plugin deployment diagram

Figure 45 shows the deployment diagram of the Eclipse domain plugin in the computer of the end user. The TFG Domain core product will be installed as well, and it will use the java compiler installed in the computer. The Eclipse workbench the developer uses will be extended by the Eclipse domain plugin to add the new features to load the model and perform the different conversions offered by the project.

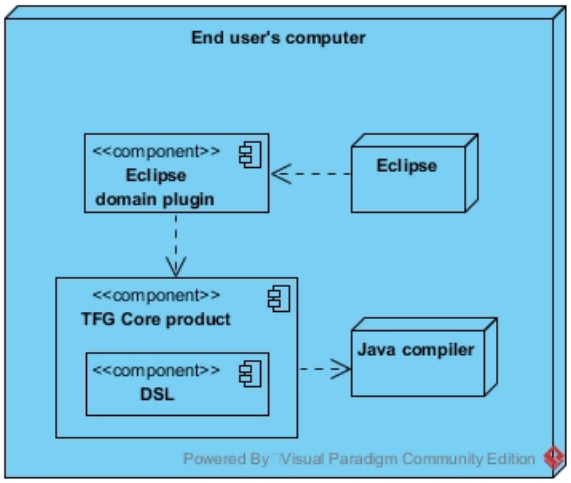

Figure 45. Eclipse domain plugin deployment diagram

3.2.2.3. Maven plugin deployment diagram

Figure 46 shows the deployment diagram of the Maven plugin in the computer of the end user. The TFG Domain core product will be installed as well, and it will use the java compiler installed in the computer. When a pom.xml Maven plugin is configured so that it uses the Maven plugin and Maven executes that file, the Maven plugin will also be executed.

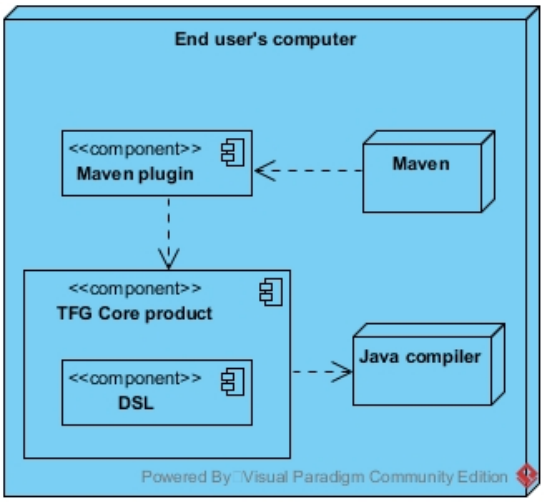

Figure 46. Maven plugin deployment diagram

3.2.2.4. Eclipse DSL plugin deployment diagram

Figure 47 shows the deployment diagram of the Eclipse DSL plugin in the computer of the end user. The TFG Domain core product will not be installed, but the DSL product integrated in it will be. When the developer or domain expert installs the plugin in their Eclipse workbench, the Eclipse environment will be extended to offer the DSL editor along with its features, such as autocomplete, validations of syntax and compilation errors.

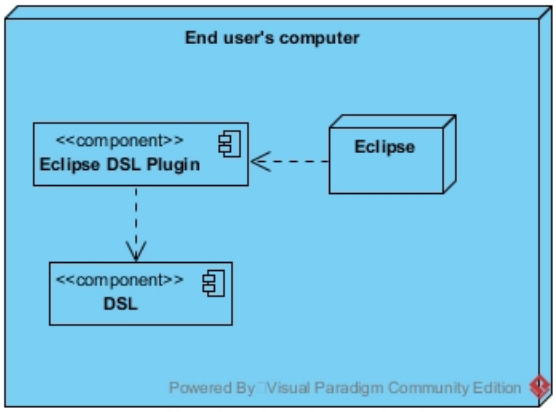

Figure 47. Eclipse DSL plugin deployment diagram

3.2.2.5. Eclipse product deployment diagram

Figure 48 shows the deployment diagram of the Eclipse product in the computer of the end user. The TFG Domain core product will not be installed, but the DSL product integrated in it will be. Moreover, features related to the Eclipse platform have been added and integrated with the Eclipse product so that it can launch a new Eclipse workbench. When the developers or domain experts execute the product, a new Eclipse workbench will be launched, with the DSL product already integrated and installed, so the DSL editor is already available.

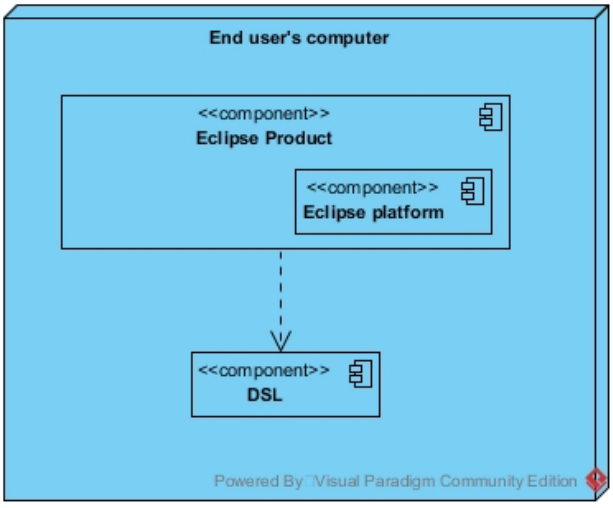

Figure 48. Eclipse product deployment diagram

3.2.3. Component diagram

A component diagram is a visual representation of the different components and the relationships between them that make up the whole system. A component is a software element formed by other elements such as classes and other components. They can be seen as logical units with a purpose, and should have only one concern. Therefore, a component diagram is a higher level abstraction grouping of a class diagram.

3.2.3.1. TFG Domain core product

The component diagram of the TFG Domain core product shows the components and the relationships that form the product. Shows the diagram.

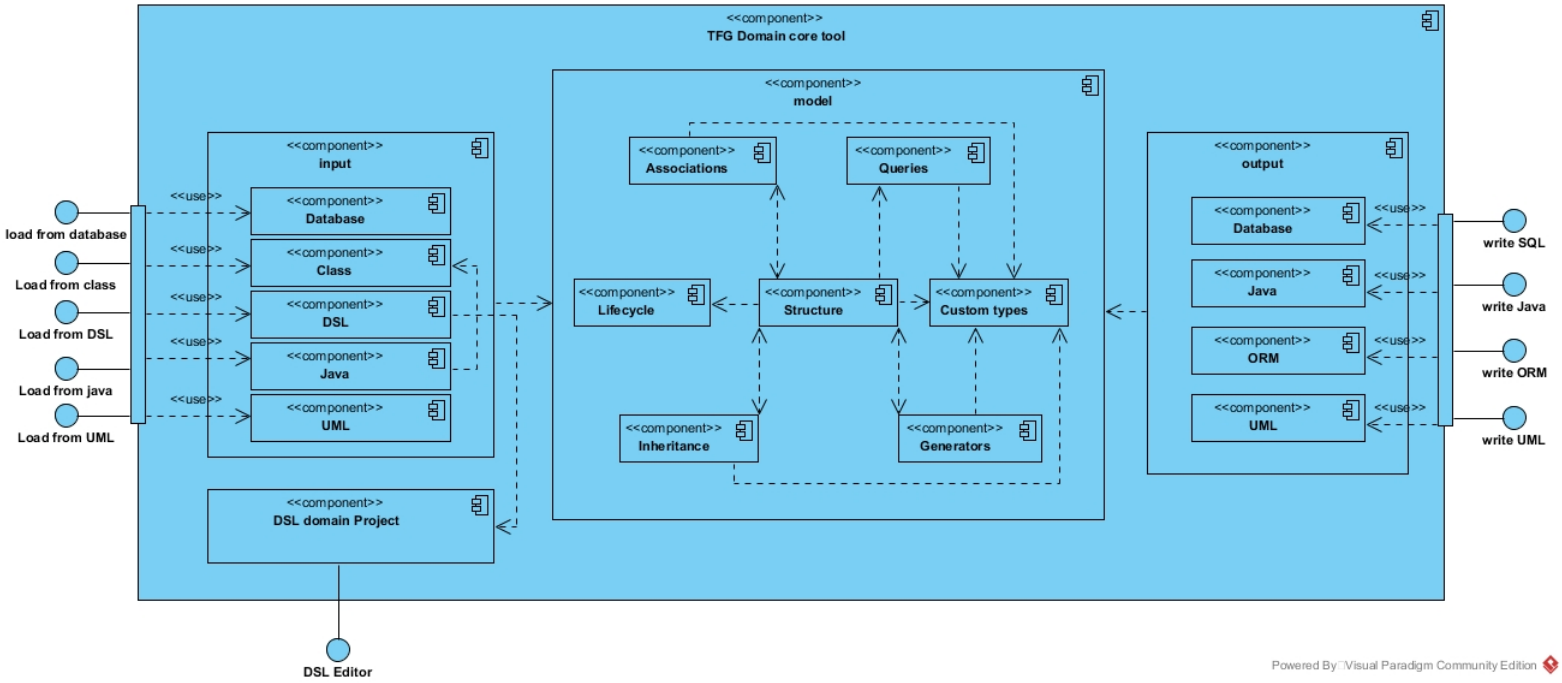

Figure 49. TFG Domain core tool component diagram

- The **input** component is in charge of loading the model from different formats or technologies. When the model is loaded, an intermediate model is created expressing the essence of the domain model.
    - The **database** component is in charge of connecting to the specified database with the connection parameters provided, analyzing the structure of the database and creating the intermediate model.
    - The **class** component is in charge of loading the compiled classes with the domain, analyzing them and creating the intermediate model.
    - The **DSL** component is in charge of loading the specified .tfg DSL file, compile it and from the semantic model obtained, generate the intermediate model.
    - The **java** component is in charge of compiling the specified java files, and then passing the compiled files to the **class** component so that it loads the model and generates the intermediate model.
    - The **UML** component is in charge of loading the specified UML model file, and then traverse and analyze the different elements to generate the intermediate model.
- The **model** component groups and contains the domain model of this project. It contains elements that represent metadata or elements that form or express other domain models.
    - The **associations** component contains elements used to configure and specify associations between entities.
    - The **queries** component contains elements used to configure and generate different queries to manipulate and retrieve data from the database.

    - The **lifecycle** component contains elements used to configure and establish different functions or procedures to execute at different stages in the lifecycle of entities.
    - The **inheritance** component is used to configure how inheritance in the java domain model will be mapped in the database.
    - The **generators** component is used to configure and create elements in charge of generating automatically values for the identity of entities.
    - The **custom types** component groups custom types of the domain model.
    - The **structure** component contains core elements to express the essence of any domain model, such as entities, value objects, elements that represent custom types, and the package structure.
- The **output** component is in charge of performing the different conversions to generate the domain model in the desired form.
    - The **database** component is in charge of creating the database script necessary to create the database structure with tables and relations between them to express the domain model.
    - The **java** component is in charge of creating the java classes that represent the domain model in the source code, and an extra additional class to handle the associations.
    - The **ORM** component is in charge of generating the ORM persistence configuration file to be used by the persistence provider and map the Object-Oriented model with the relational database structure, and vice versa. If the original source code is available, it also duplicates these source code files and removes any persistence configuration in them, to achieve a better model, better software and follow good software practices.
    - The **UML** component is in charge of generating a UML model file from the intermediate domain model loaded, and also produce a UML file containing the UML profile used in the UML model file.

3.2.3.2. DSL domain project

While inner components of the TFG Domain core tool are pretty simple and straightforward, so a component diagram would not provide any benefit, the DSL domain project is a bit more complex and composed by several inner projects. Therefore, an additional component diagram is provided in this section.

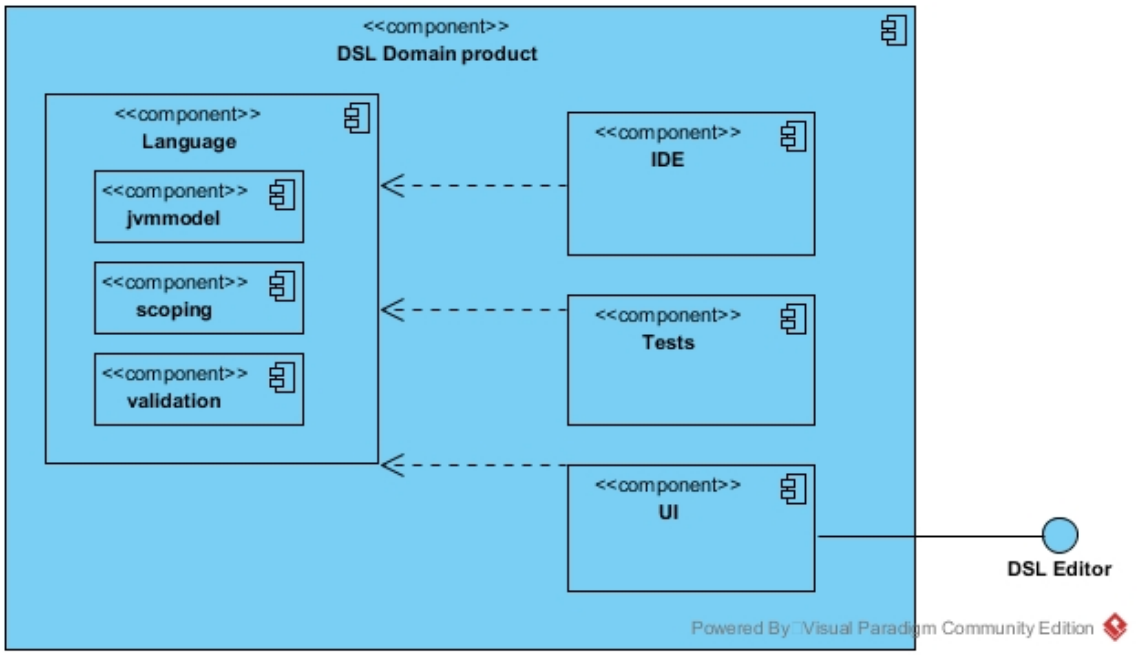

Figure 50. DSL Domain product component diagram

- The **language** component contains the language and its grammar defined for the project, and other additional elements related with the language such as the semantic model, custom validators, lexer, linker and parser, etc.).
    - The **jvmmodel** component contains an element called *JvmModelInferrer*. Once the DSL model is defined, an element must be created to infer java elements and objects from the different DSL elements written in the DSL file.
    - The **scoping** component contains an element that may be used to customize the scope of elements.
    - The **validation** component contains elements that may be used to create custom validations that can be integrated in the DSL language.
- The **IDE** component contains elements that offer services characteristic of *IDE (Integrated Development Environment*). That is, no matter the platform, they offer services and features for content assist that can be integrated in programs developers use to write source code.
- The **tests** component contains elements that can be used to create tests and check the behavior and correctness of the DSL project.
- The **UI** component offers services related to the Eclipse workbench like the DSL editor.

3.2.3.3. Other components

The different clients offered with the project are pretty simple and straightforward since they are in charge of user interaction, and then they delegate the business logic to the TFG Domain core product. Moreover, sometimes the important aspect to produce another client lies within configuration files and exporting options. However, component diagrams of three of the products are presented so the reader can see their structure and how simple they are. The other two products (the Eclipse DSL plugin and the Eclipse product) are basically configurations and exportations of the DSL project integrated in the TFG Domain core product.

3.2.3.3.1. Maven domain plugin

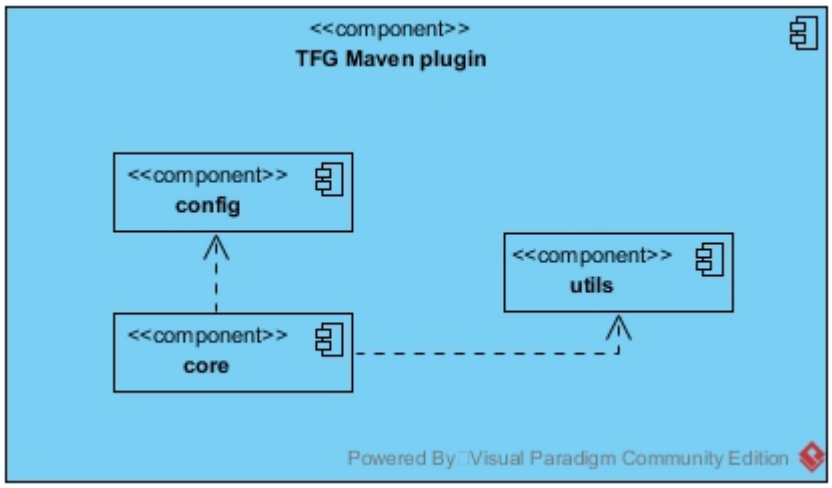

Figure 51. Component diagram of Maven plugin

- The **core** component contains the java class that creates the plugin and executes the different services the client of the plugin wants to execute.
- The **utils** component offers utility services to the **core** component to ease its work, such as obtaining the right reader and writer for the domain model and the conversions.
- The **config** component contains the elements that create the configuration the client of the plugin uses to tell the plugin what model to read and load, and what conversions to apply to the model loaded.

3.2.3.3.2. Eclipse domain plugin

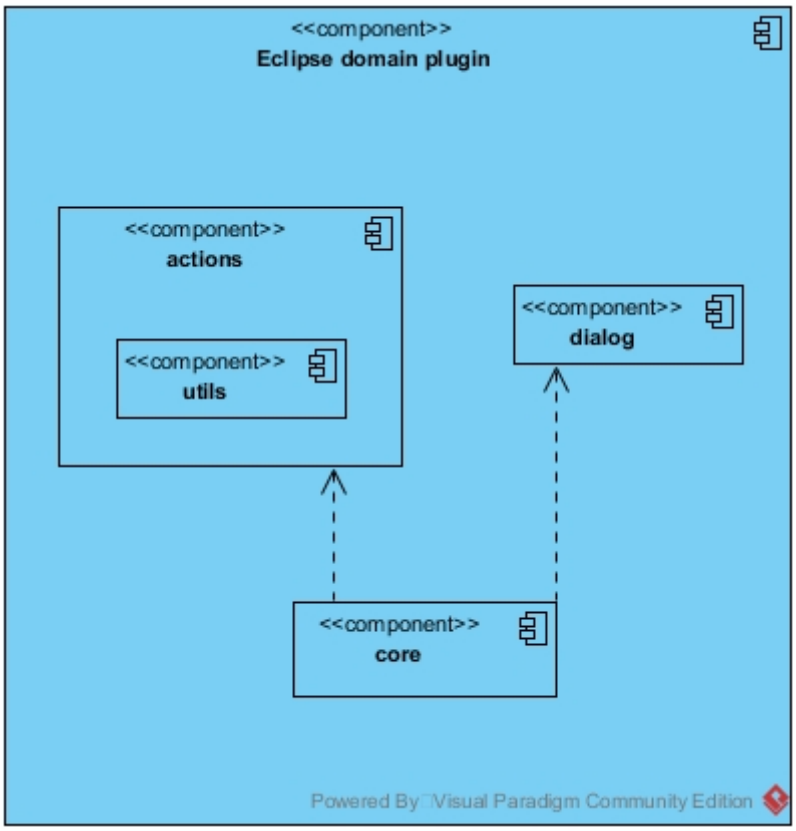

Figure 52. Component diagram of Eclipse domain plugin

- The **core** component contains configuration files that indicates how the plugin will be integrated in the Eclipse workbench, and how it will extend it. Moreover, it also contains classes to activate the plugin and to store the domain model loaded.
- The **dialog** component creates a custom dialog used so that the user can introduce configuration parameters to connect to a database and read the model from that database.
- The **actions** component contains software elements that, depending on the Eclipse plugin menu selected by the user, will call the different services offered by the TFG Domain core product.

3.2.3.3.3. Command-line program

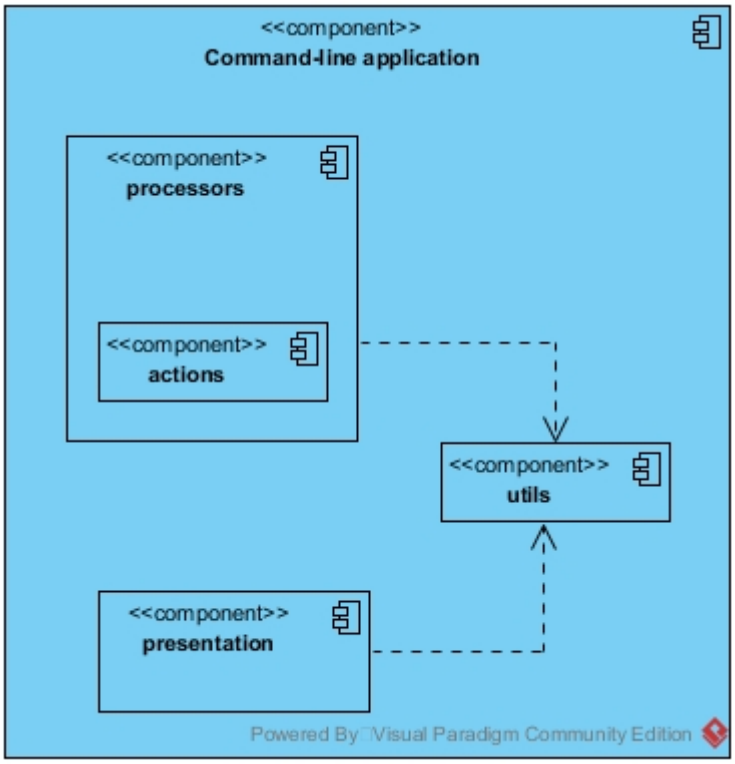

Figure 53. Component diagram of the command-line application

- The **presentation** component is in charge of displaying the initial welcome message, read commands from the user and then redirect to the corresponding processor depending on the command to be executed.
- The **utils** component offers utility services to help with the execution of the program, doing tasks such as accessing a file with all the messages the application displays, break down the user input into the different options and parameters readable by the program, and checking the number of input options is correct.
- The **processors** component contains the software elements that, depending on the command and options, will make different calls to the TFG Domain core product, making different validations along the way.
    - The **actions** component contains software elements that execute the calls to the core product to load the model, and that analyze the input options passed by the user to obtain the corresponding Writer.

3.3. Package and class diagrams

In this section, package and class diagrams for the different system components of the project are presented. In a package diagram, the package structure is presented visually. A package is an organizational element used to group or organize different java elements such as java classes, interfaces, enumerations and other sub-packages. Moreover, they create

namespaces, so that a java class with the name *Dog* in the package *pckgA* is not the same java as another java named *Dog*, but located in package *packgB*. If it helps the reader, a package could be thought as a computer folder, containing files and other folders.

A class diagram displays the different classes in a component or module, their attributes and behavior, and their dependencies to other classes.

3.3.1. Package diagrams

3.3.1.1. TFG Domain core product package diagram

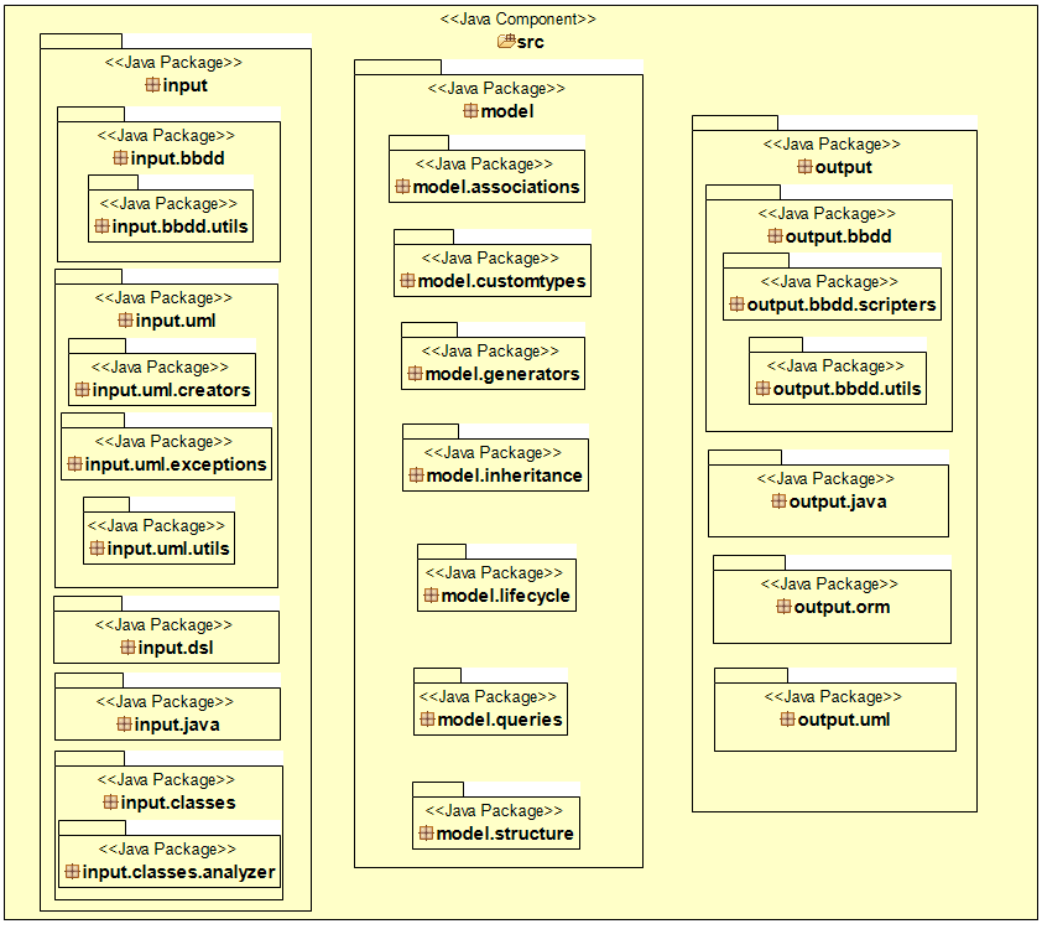

Figure 54. TFG Domain core package diagram

Package name	Description
input	Contains software modules and elements to load a domain model.
model	Contains the different java classes that express the domain model of the project, allowing to express data about other domains.
output	Contains software modules and elements to perform different conversions and generate new domain models in different forms.

input.bbdd	Contains software modules and elements that connect to a database, read and analyze its metadata, and load the model from the database.
input.bbdd.utils	Contains software elements that offer utility and supporting services for other elements performing the task of loading the domain model from the database.
input.uml	Contains software modules and elements that load a UML model file, parse it, analyze its content and generate the intermediate model from it.
input.uml.creators	Contains software elements that handle the creation of different domain model elements in the intermediate model of the tool from the UML model file.
input.uml.exceptions	Contains custom errors that may arise when loading the model from a UML model file.
input.uml.utils	Contains software elements that offer utility and supporting services for other elements performing the task of loading the domain model from the UML model file.
input.dsl	Contains software module and elements that load a DSL file, parse it and from the semantic model generate the intermediate domain model of the project.
input.java	Contains software module and elements that load java files, compile them and delegates the loading operation to the input.classes package.
input.classes	Contains software module and elements that load java compiled classes, analyze them to find persistence configurations, and generate the intermediate model of the project.
input.classes.analyzer	Contains software elements that analyze the different persistence configurations found in the loaded compiled java classes to complete and generate the intermediate model.
model.association	Contains elements used to configure and represent associations in other domain models.
model.customtypes	Contains custom types used to restrict the values of attributes and configurations in the domain model.
model.generators	Contains elements used to configure and represent different identity generators in the domain models.
model.inheritance	Contains elements used to configure and represent how inheritance is mapped in the relational database in domain models.
model.lifecycle	Contains elements used to configure and establish different methods and procedures to be executed at different stages in the lifecycle of domain model elements.
model.queries	Contains elements used to create and configure database queries to retrieve and manipulate data in domain models.
model.structure	Contains elements used to indicate and represent the structure of domain models loaded in the project.
output.bbdd	Contains software module and elements that traverse the domain model elements, performing conversions and generating database scripts to create tables and relationships.
output.bbdd.scripters	Contains software elements in charge of generating specific pieces of SQL scripts to create tables and/or relationships.

output.bbdd.utils	Contains software elements that offer utility and supporting services to other software components in the output.bbdd package.
output.java	Contains software module and elements that traverse the domain model elements, performing conversions and generating the different java source code files representing the loaded domain model.
output.orm	Contains software module and elements that traverse the domain model elements, performing conversions and generating the ORM persistence configuration file of the loaded domain model.
output.uml	Contains software module and elements that traverse the domain model elements, performing conversions and generating the UML model file representing with UML elements the components of the loaded domain model.

Table 2. Table of packages overview in the TFG Domain core product

3.3.1.2. Command-line application package diagram

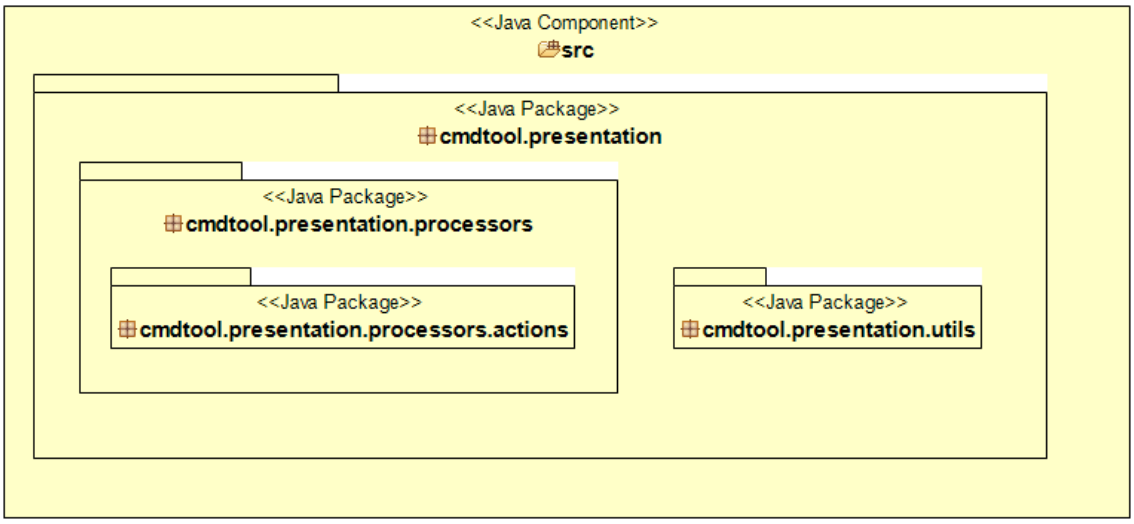

Figure 55. Command-line application package diagram

Package name	Description
cmdtool.presentation	Contains all the components and elements that form the command-line application.
cmdtool.presentation.processors	Contains the different software elements in charge of executing the different commands specified by the user and supported by the tool.
cmdtool.presentation.processors.actions	Contains the different software elements that call the corresponding elements in the TFG Domain core tool to load the model and to get the writer to perform the conversions.
cmdtool.presentation.utils	Contains software elements that offer supporting and utility services to other elements in the command-line project.

Table 3. Table of packages overview in the Command-line application

3.3.1.3. Eclipse domain plugin package diagram

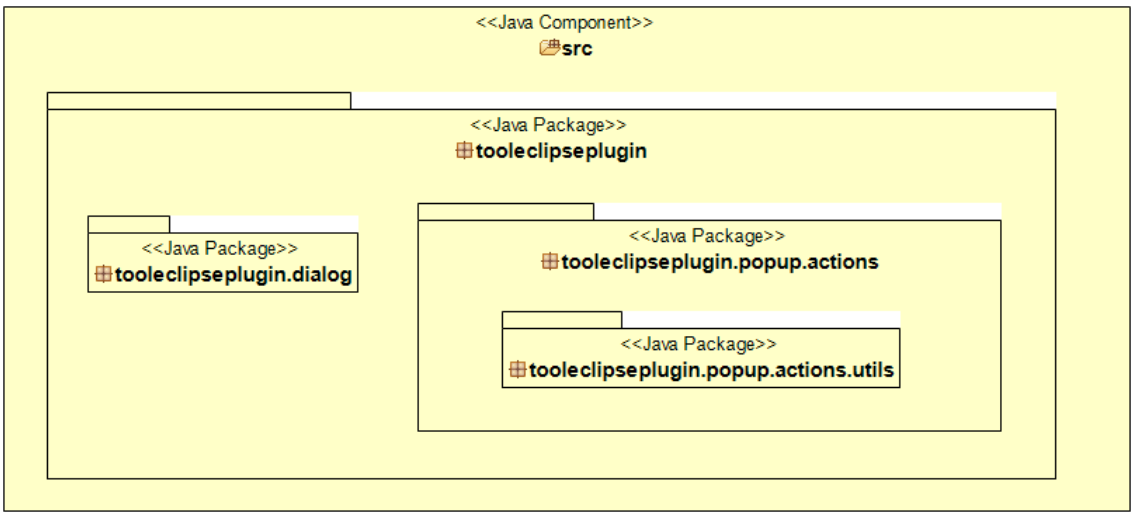

Figure 56. Eclipse domain plugin package diagram

Package name	Description
tooleclipseplugin	Contains the software components that form the Eclipse domain plugin.
tooleclipseplugin.dialog	Contains the software element in charge of creating the custom Eclipse dialog window to ask the user for database connection configurations, and store the input data.
tooleclipseplugin.popup.actions	Contains the software elements in charge of executing the different tasks or commands specified by the user through the popup options of this plugin.
tooleclipseplugin.popup.actions.utils	Contains software elements that offer supporting and utility services to other software components in the Eclipse domain plugin project.

Table 4. Table of packages overview in the Eclipse domain plugin

3.3.1.4. Maven plugin package diagram

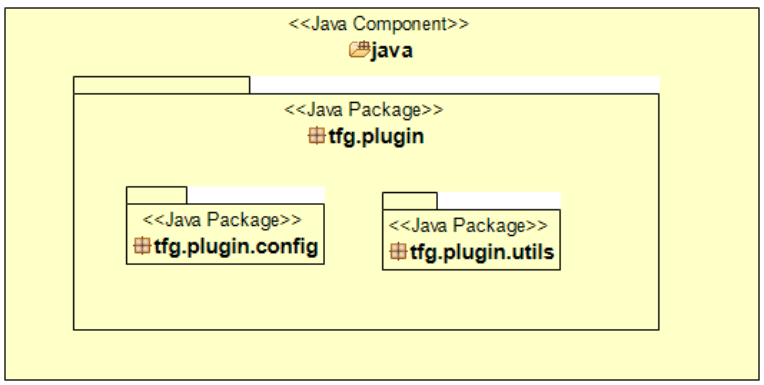

Figure 57. Maven plugin package diagram

Package name	Description
tfg.plugin	Contains the software components that form the Maven plugin.
tfg.plugin.config	Contains the software elements used to specify the Maven tags (tags are XML elements enclosed in the "<" and ">" symbols) the user can use to execute the different services provided by this plugin.
tfg.plugin.utils	Contains software elements that provide different utility and supporting operations to other components in the Maven plugin project.

Table 5. Table of packages overview in the Maven plugin

3.3.1.5. DSL parent package diagram

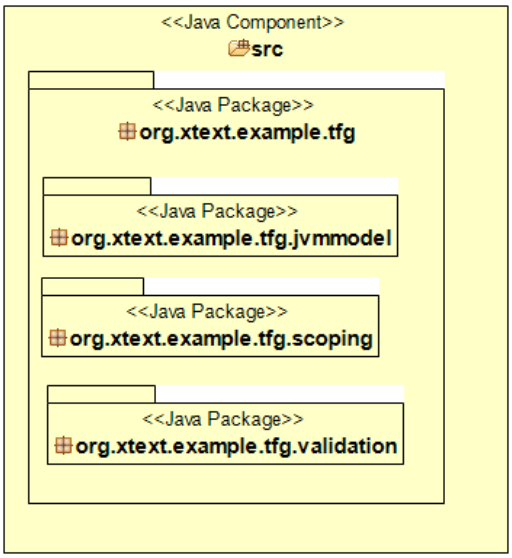

Figure 58. DSL parent package diagram

Package name	Description
org.xtext.example.tfg	Contains elements that form the DSL parent project, creating generation DSL artifacts and other components related to the language and its grammar.
org.xtext.example.tfg.jvmmodel	Contains software elements that infer java elements and objects from the different DSL elements written in the DSL file.
org.xtext.example.tfg.scoping	Contains software elements that may be use to customize the scope of elements in the DSL file.
org.xtext.example.tfg.validation	Contains elements that may be used to create custom validations that can be integrated in the DSL language.

Table 6. Table of packages overview in the DSL parent project

3.3.1.6. DSL UI package diagram

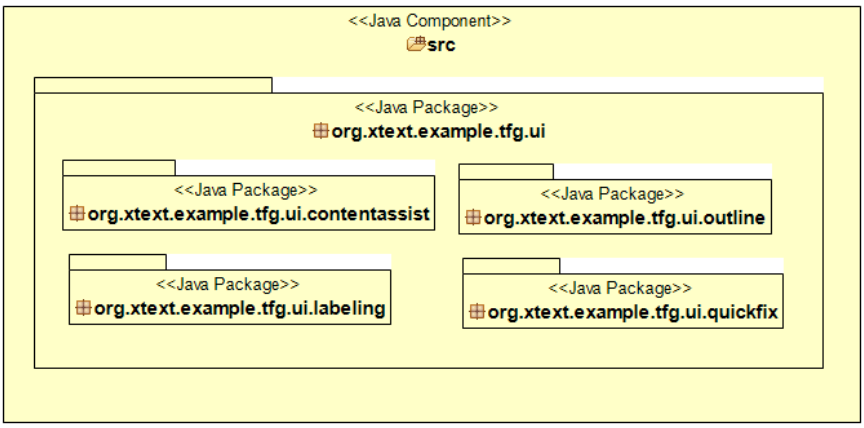

Figure 59. DSL UI project package diagram

Package name	Description
org.xtext.example.tfg.ui	Contains components that form the DSL UI project.
org.xtext.example.tfg.ui.contentassist	Contains elements that implement the content assist feature of the DSL editor. Content assist is a feature that offers options such as autocomplete which, depending on the context, suggest different language elements. This increases development productivity.
org.xtext.example.tfg.ui.labeling	Contains elements that focus on displaying text, messages and icons in different parts and elements of the DSL editor.
org.xtext.example.tfg.ui.outline	Contains elements that create and handle the outline view. This view is a graphic element displayed in the IDE that shows the DSL elements created with the DSL, and their structure.
org.xtext.example.tfg.ui.quickfix	Contains elements that provide quick fixes. In IDEs, when an error is raised and displayed to the user through the editor, some quick solutions for the error might be suggested. These elements handle that feature.

Table 7. Table of packages overview in the DSL UI project

3.3.1.7. Package diagram of other products

The other products of the project, the Eclipse DSL plugin and the Eclipse product, are created using different configurations and exportation options of the DSL product integrated in the TFG Domain core product. Therefore, no package diagrams are provided for them.

Moreover, subprojects of the DSL project such as the tests or IDE ones are empty components automatically created by XText that can be used in the future to extend this tool and add tests and domain validations into the DSL language, but since those features are not part of the DSL at the moment, there is no point in displaying their package diagrams.

3.3.2. Class diagrams

3.3.2.1. TFG Domain core project

The TFG Domain core project contains a high number of classes, so in order to increase readability and avoid cognitive overload, different smaller class diagrams are provided for the different packages. The package structure can be seen in the package diagram.

3.3.2.1.1. Database input module

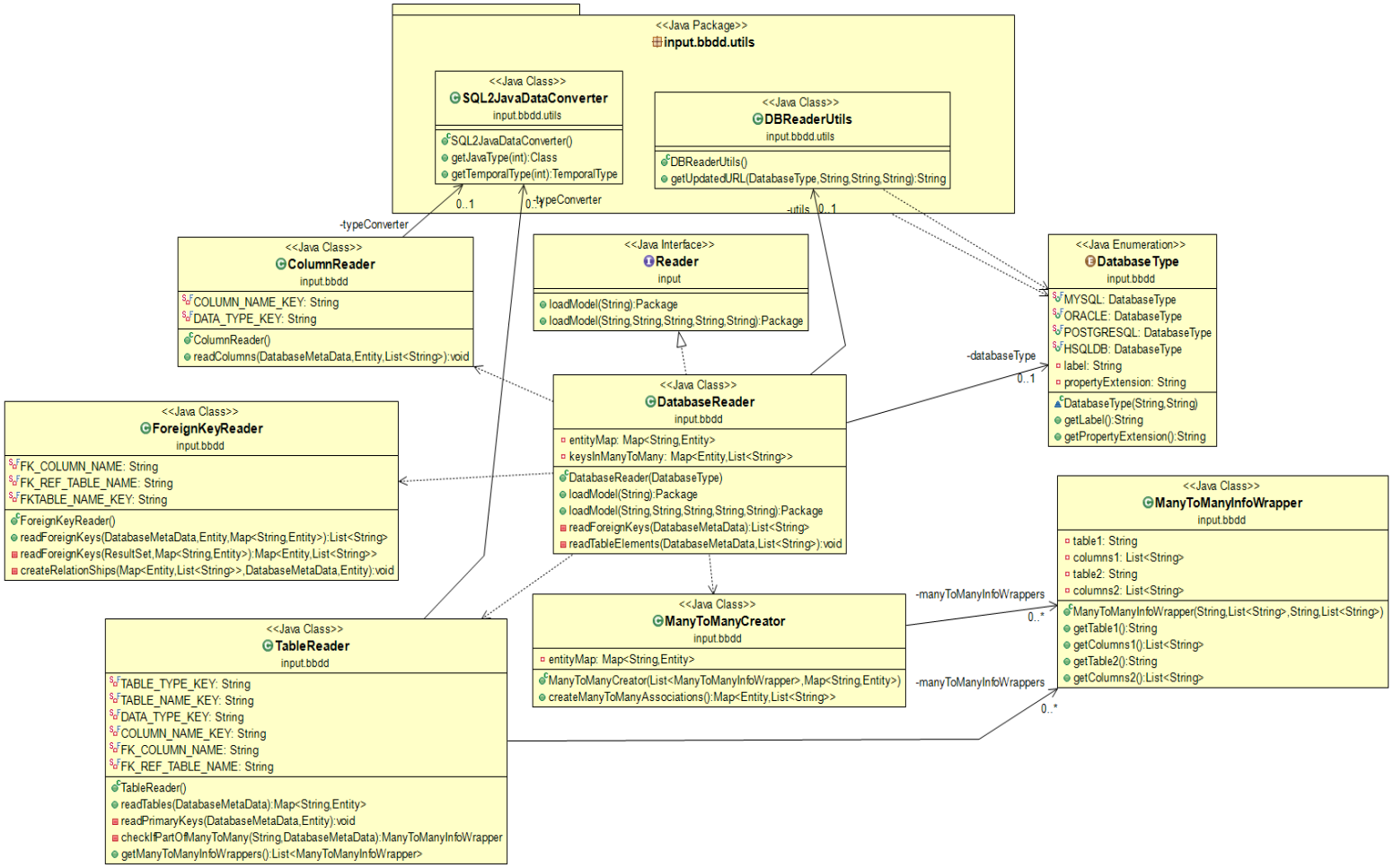

Figure 60. Database input module class diagram

3.3.2.1.2. Class input module

The class input module loads the domain model from compiled java classes. In order to analyze the different persistence configurations, different classes called “Analyzers” have been created. An analyzer class has been created to obtain persistence information about the domain model from each JPA annotation. In total, this module contains 65 analyzers in the *analyzers* package. They are very similar in form and behavior. They receive an element or persistence annotation, and they create the corresponding intermediate model object from the project’s own domain model. A high level class diagram showing just a small portion of analyzers is shown next.

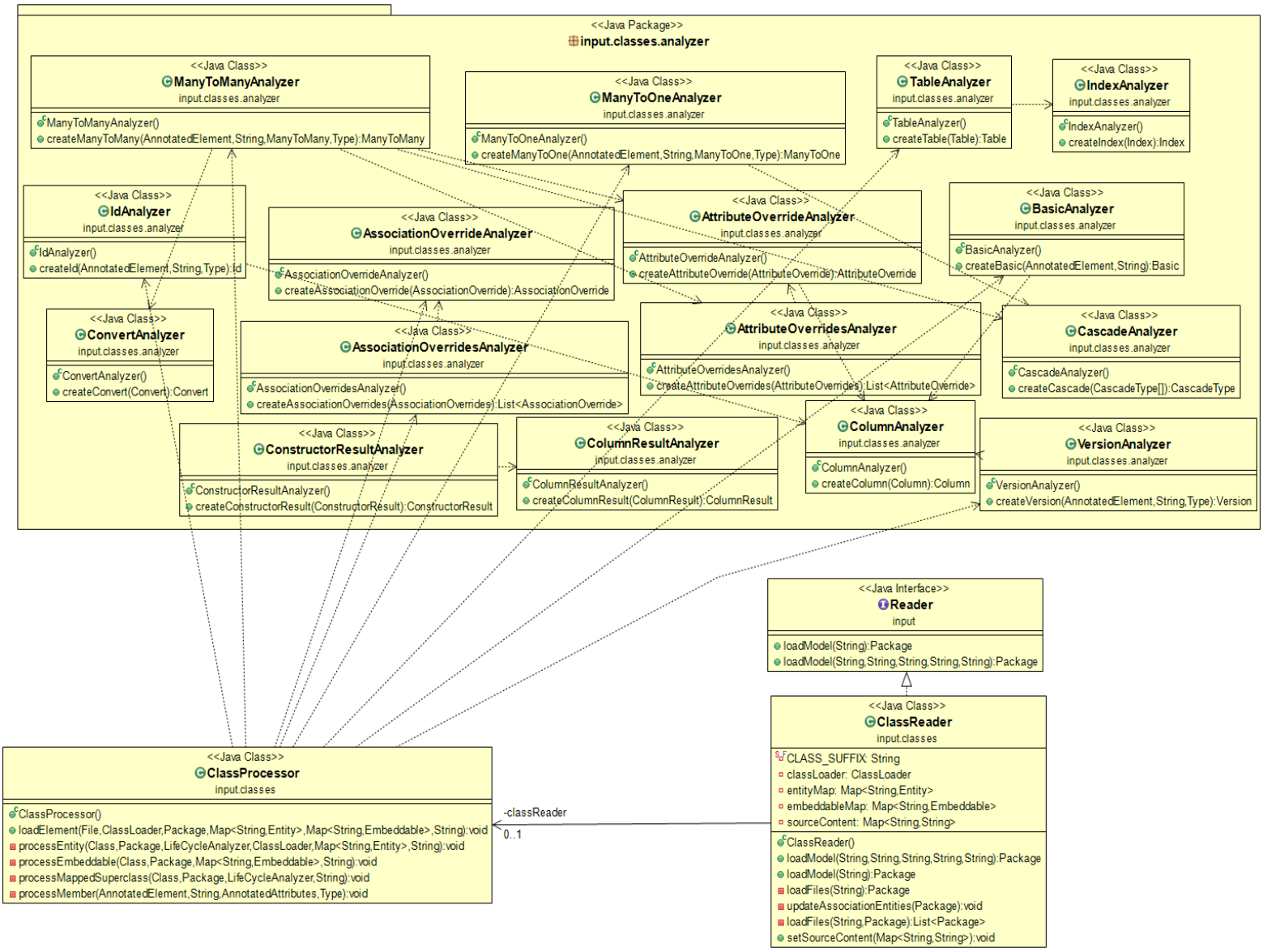

Figure 61. Class input module class diagram

3.3.2.1.3. DSL input module

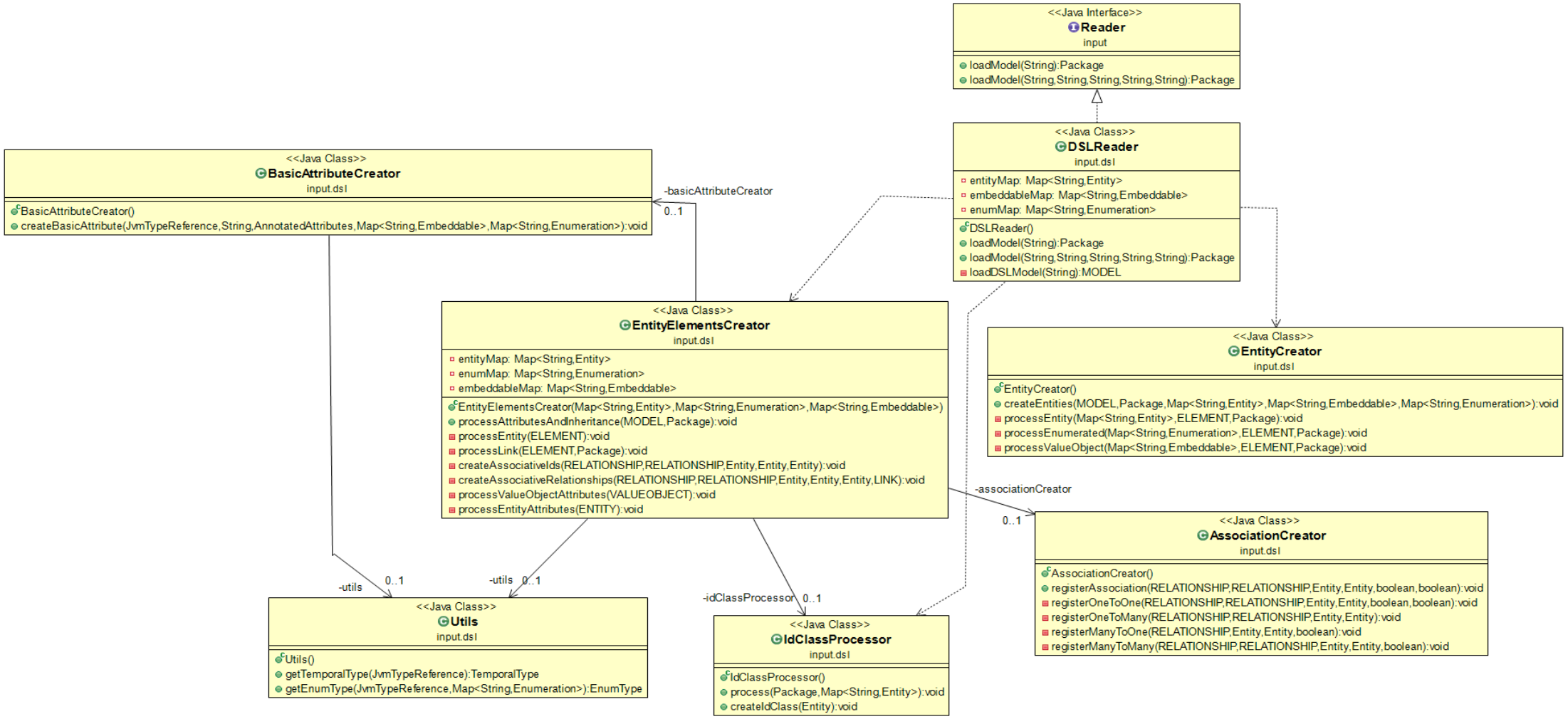

Figure 62. DSL input module class diagram

3.3.2.1.4. Java input module

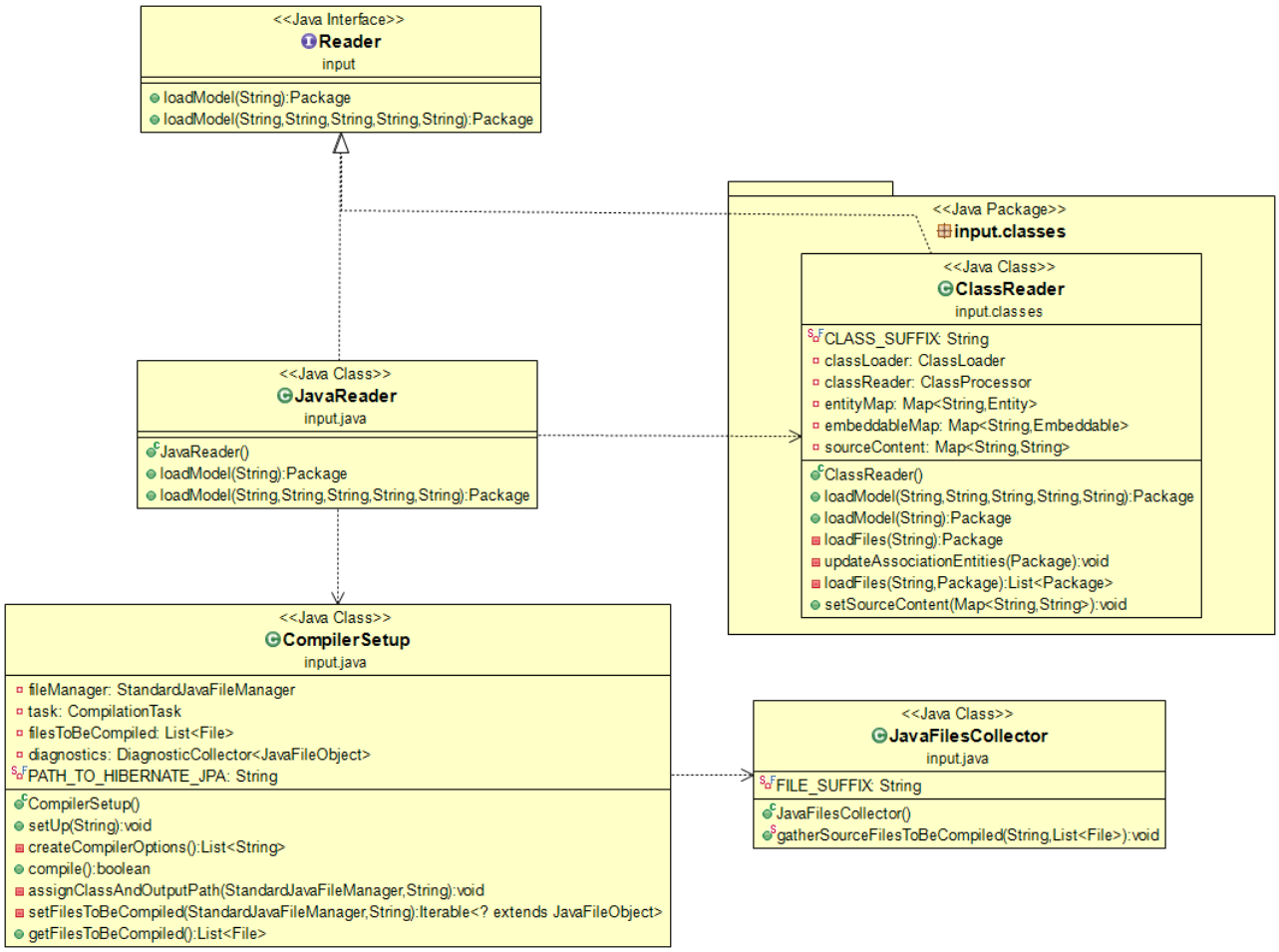

Figure 63. Java input module class diagram

3.3.2.1.5. UML input module

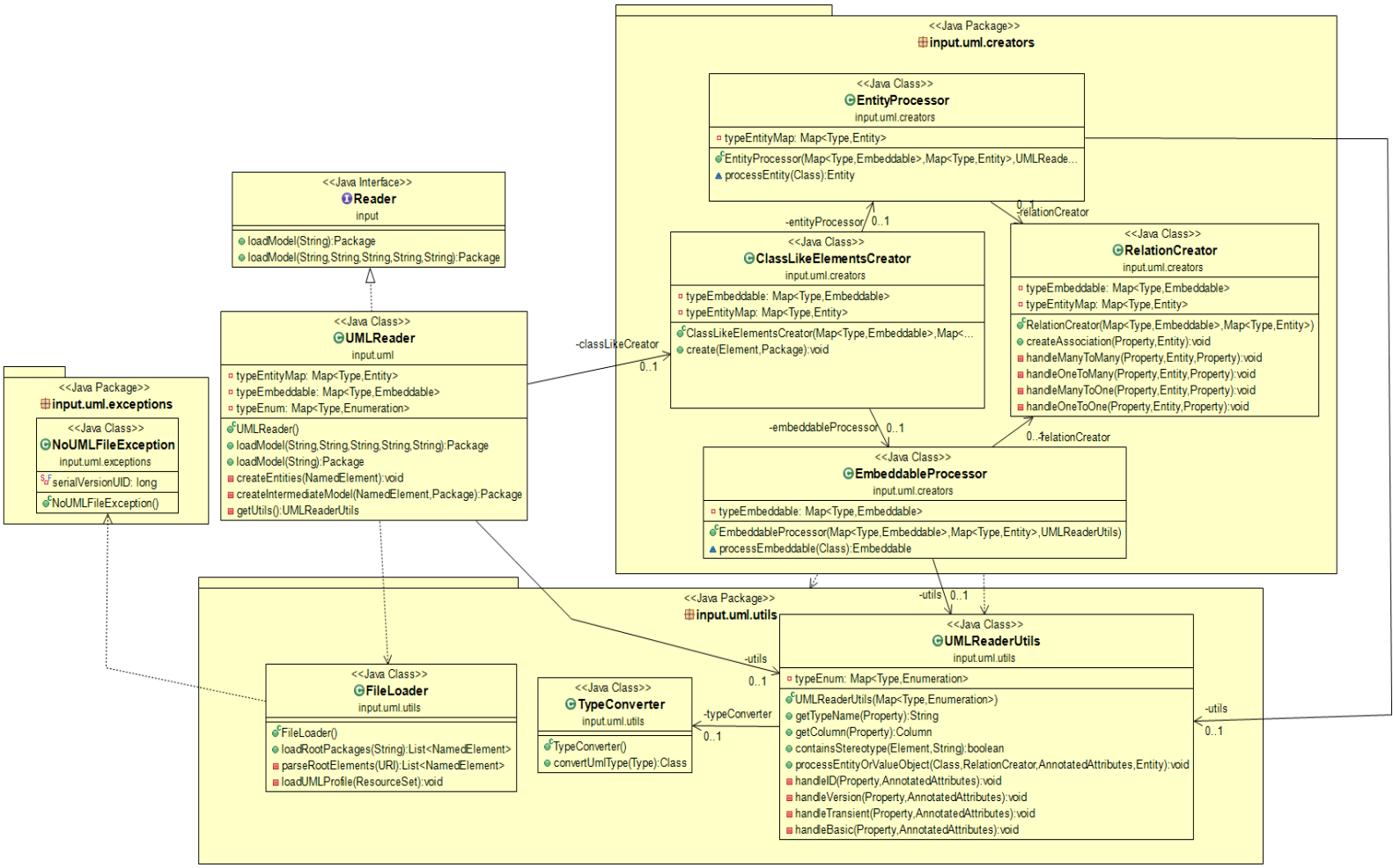

Figure 64. UML input module class diagram

3.3.2.1.6. Model component

Some diagrams of the different java classes that form the model component have been previously displayed and explained in **Error! Reference source not found.**, page **Error! Bookmark not defined.**.

3.3.2.1.7. Database output module

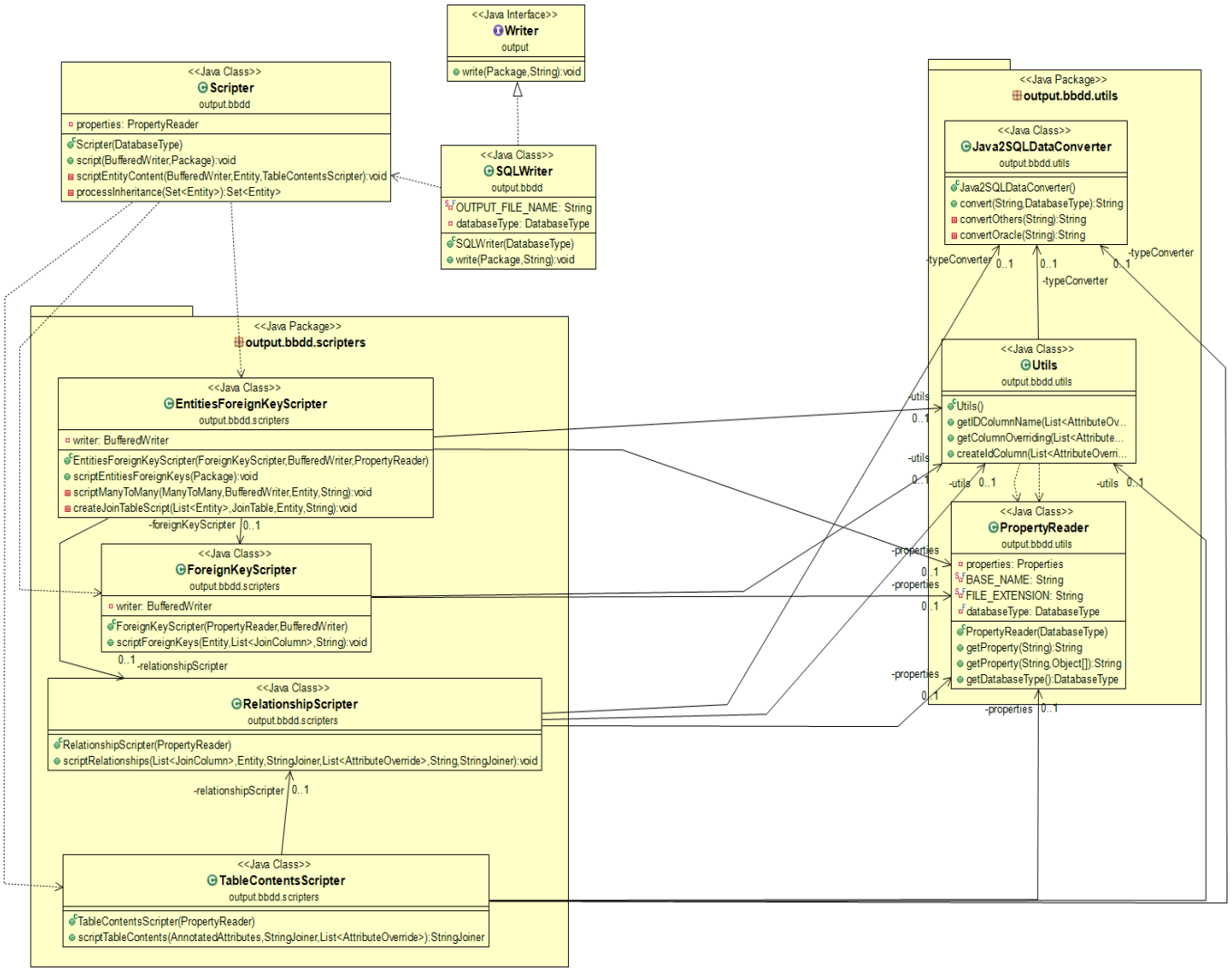

Figure 65. Database output module class diagram

3.3.2.1.8. Java output module

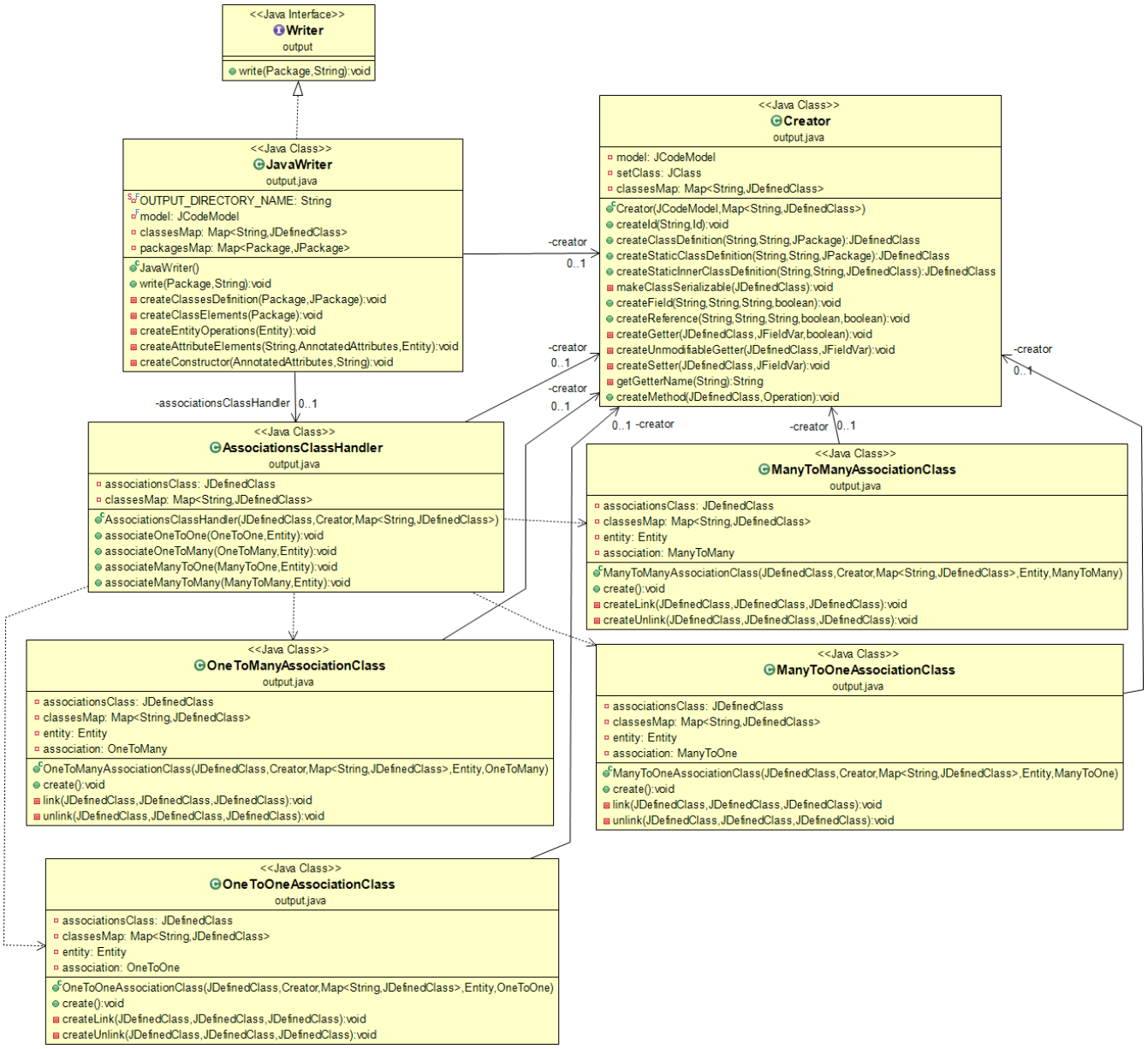

Figure 66. Java output module class diagram

3.3.2.1.9. ORM output module

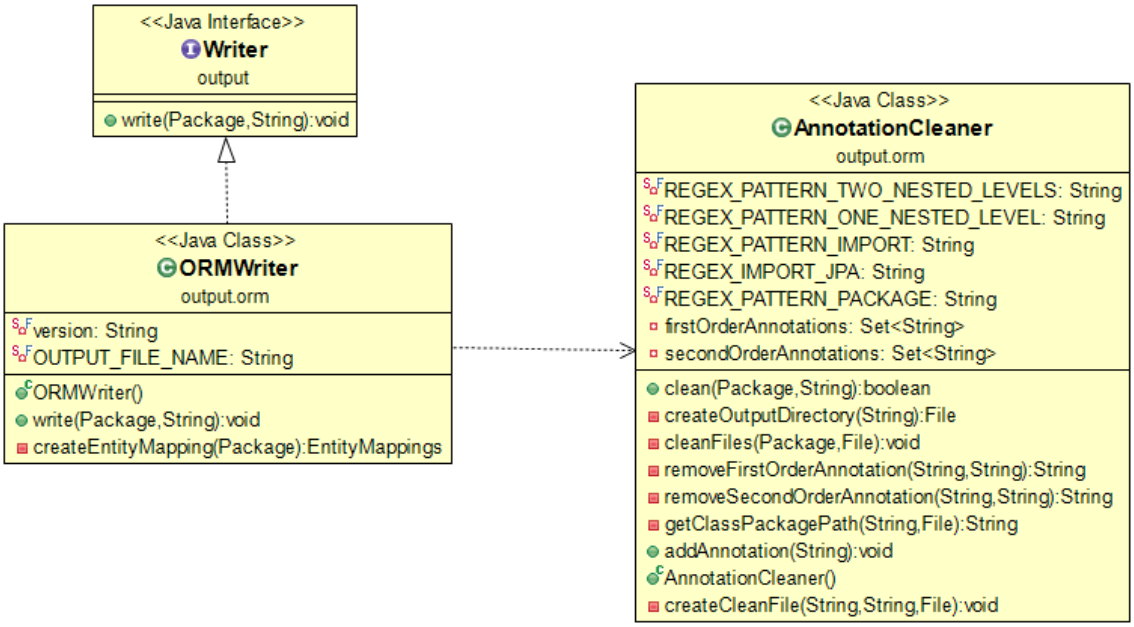

Figure 67. ORM output module class diagram

3.3.2.1.10. UML output module

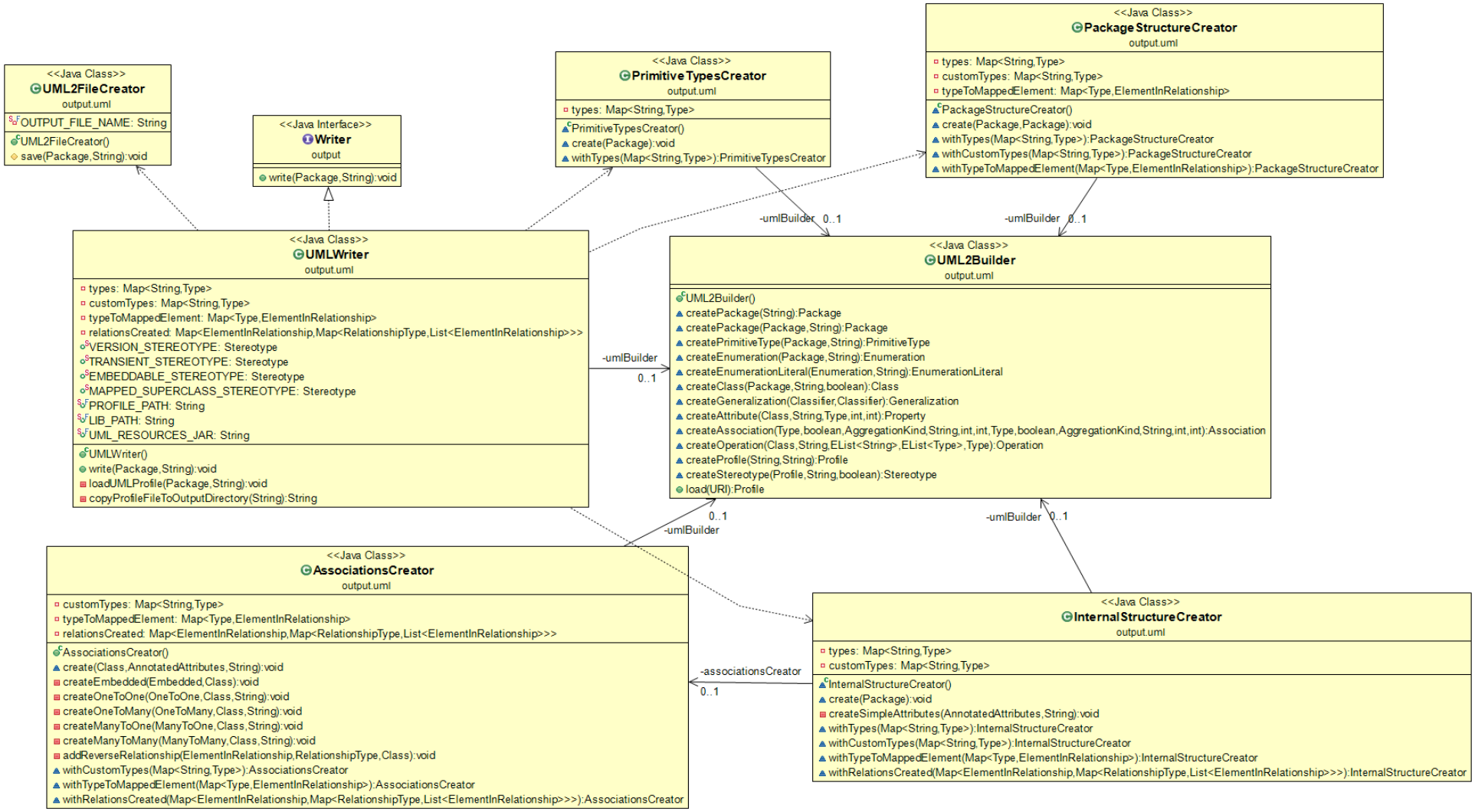

Figure 68. UML output module class diagram

3.3.2.1.11. Façades and Factories

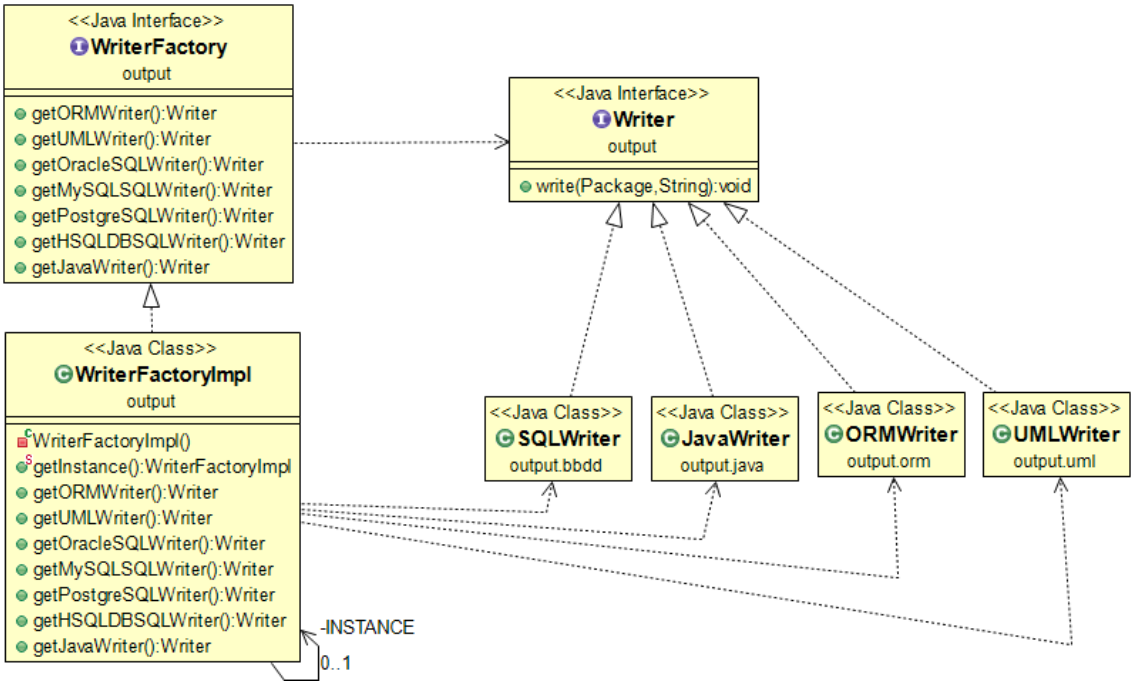

Figure 69. Writers class diagram

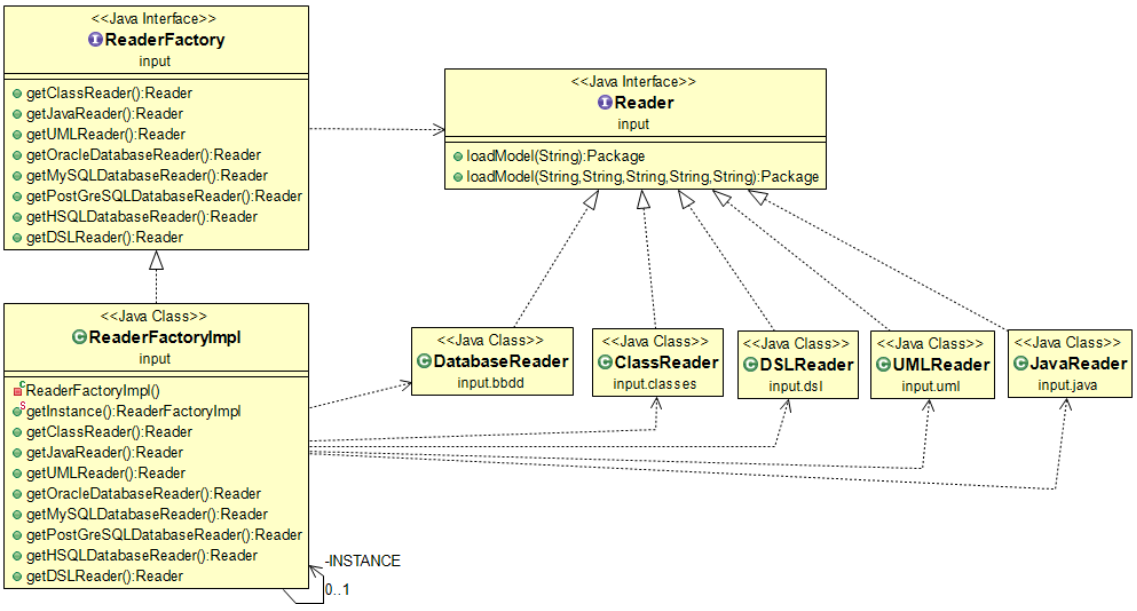

Figure 70. Readers class diagram

3.3.2.2. Command-line application

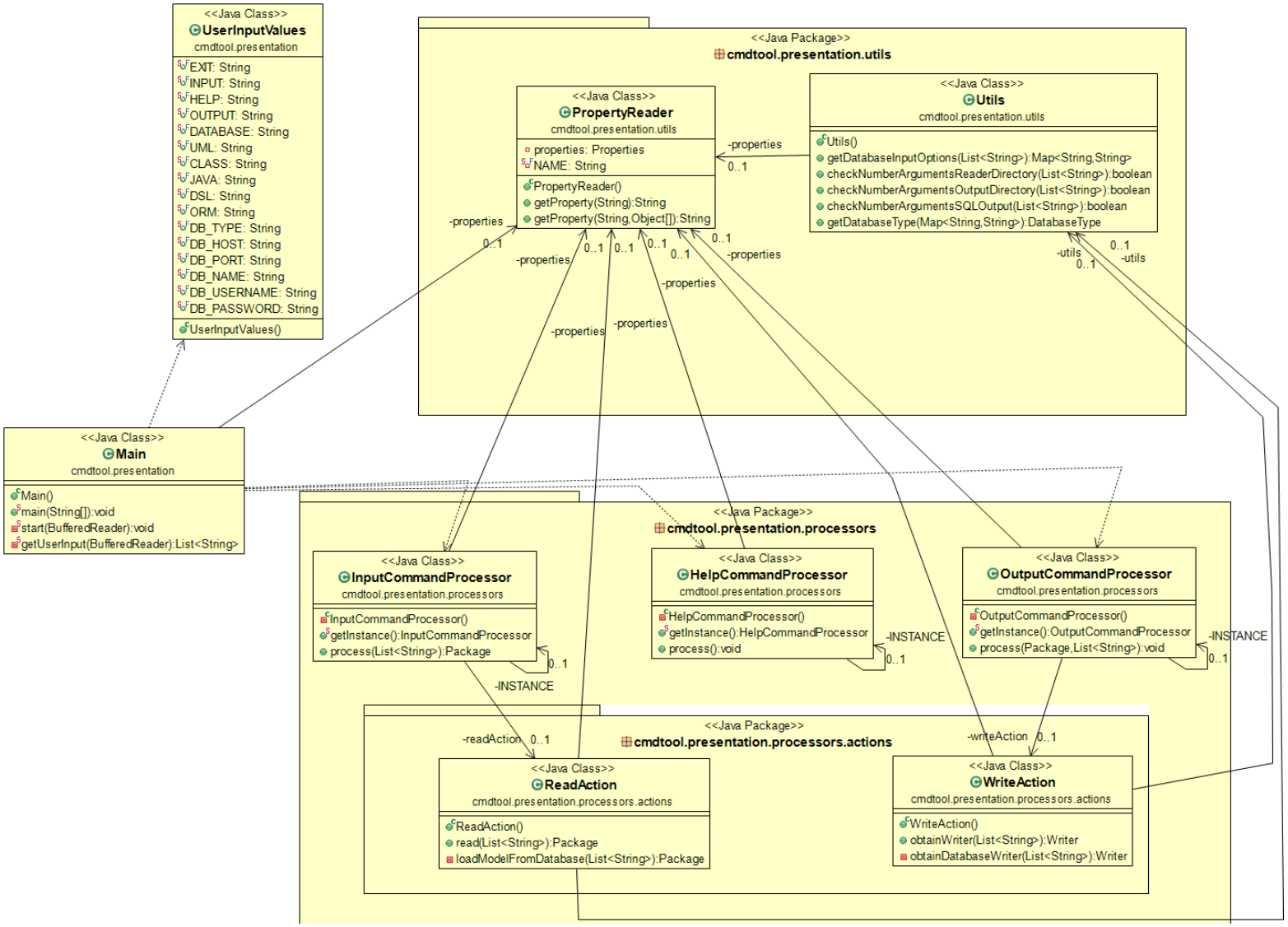

Figure 71. Command-line application class diagram

3.3.2.3. Eclipse domain plugin

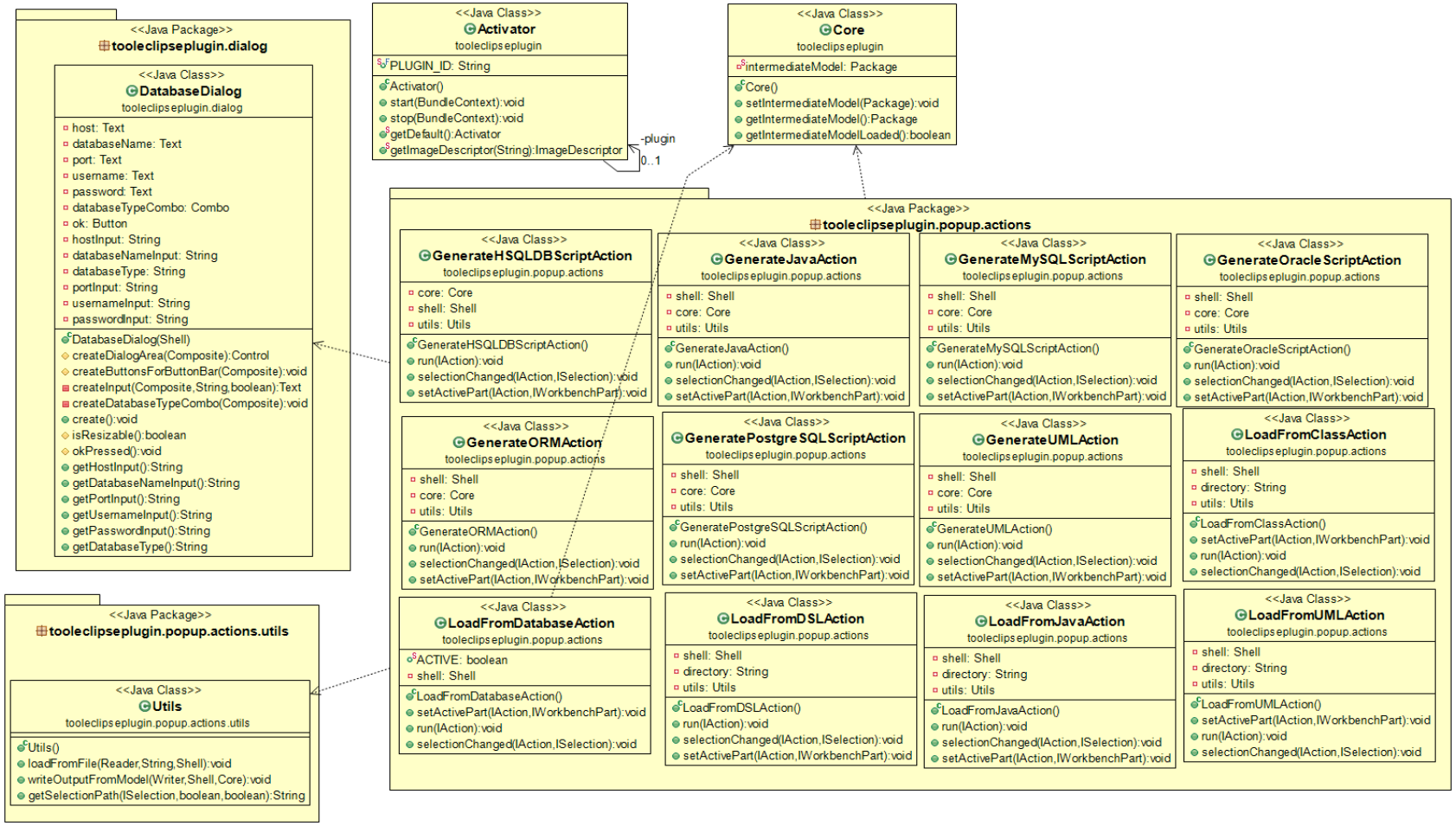

Figure 72. Eclipse domain plugin class diagram

3.3.2.4. Maven plugin

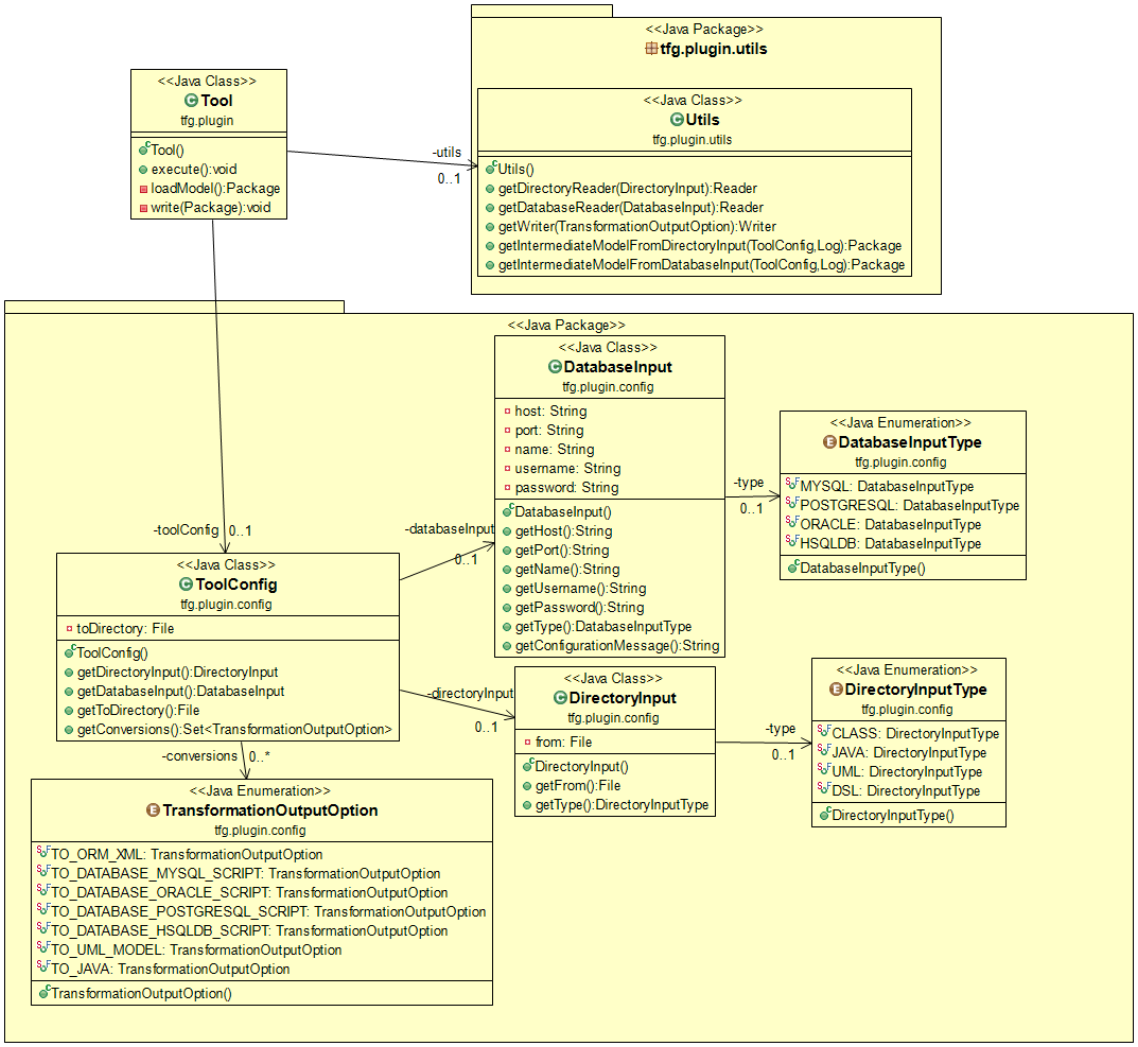

Figure 73. Maven plugin class diagram

3.3.2.5. DSL parent project

The DSL projects are a little special since they do not use the java programming language, but xtend and xtext files. Moreover, some other files and language artifacts are automatically generated as the workflow of the project is executed. However, a simple diagram is provided so that the reader can see the classes that complement the package diagram presented previously.

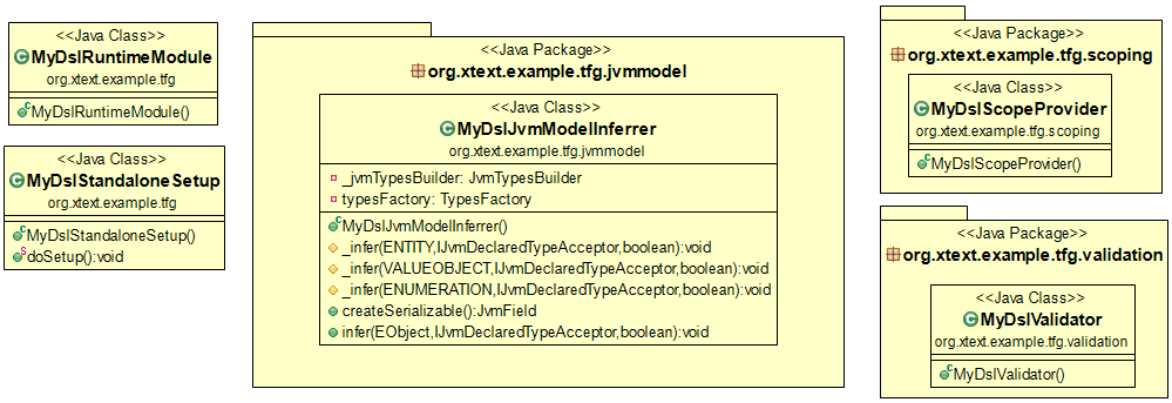

Figure 74. DSL Parent project class diagram

3.3.2.6. DSL UI project

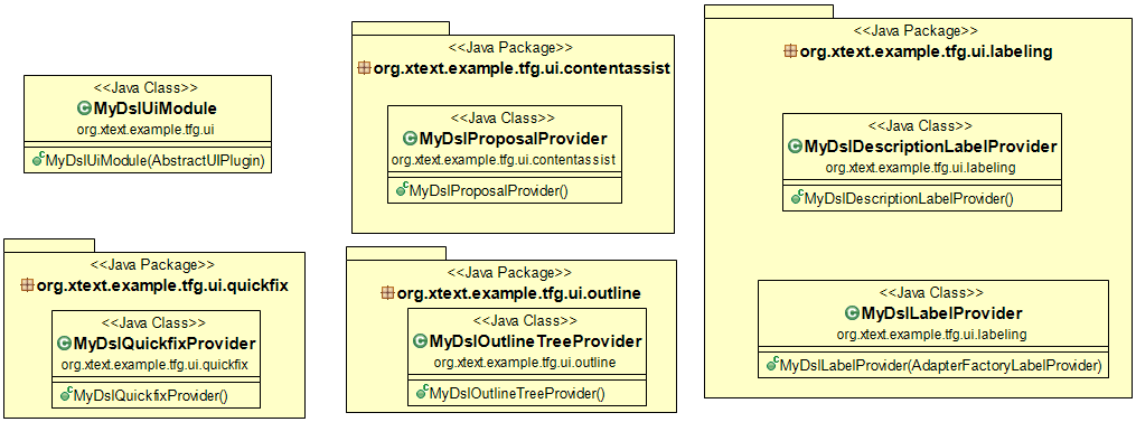

Figure 75. DSL UI project class diagram

3.3.2.7. Class diagrams of other products

The other products of the project, the Eclipse DSL plugin and the Eclipse product, are created using different configurations and exportation options of the DSL product integrated in the TFG Domain core product. Therefore, no class diagrams are provided for them.

Moreover, subprojects of the DSL project such as the tests or IDE ones are empty components automatically created by XText that can be used in the future to extend this tool and add tests and domain validations into the DSL language, but since those features are not part of the DSL at the moment, there is no point in displaying their class diagrams.

3.4. Sequence diagrams

In this next section, sequence diagrams for the different use cases identified in **Error! Reference source not found.**, page **Error! Bookmark not defined.**, are displayed so that the reader can see them with another perspective, noticing the interactions between the different software components involved in each use case.

A sequence diagram is a special type of diagram that represents visually different objects or instances of the software system, and their interactions to complete a use case. These interactions are messages or calls to methods or services offered by the elements, also called "actors". In these diagrams, another actor that may appear is the user, who may start the interaction process or use case, or other software systems. Interactions and messages between actors are organized in time sequence and numbered in the order they take place.

In the next diagrams, actors are displayed with either a person, or a rectangle. Interactions are represented with arrows, and when the arrow has a dotted or dashed line, it means that it is a response message. The lines and blue thin rectangles below the actors are their life lines, that is, when these objects are created and participate in the interactions.

Finally, there are two other elements that appear in the diagrams: the loop and alternative blocks. The loop block is a rectangle with the *loop* keyword at the top left corner and it means that everything inside the block will be executed repeatedly over a sequence or collection (a loop). The alternative block is a rectangle with the *alt* keyword at the top left corner, and a dashed line separating two parts of the block. It means that, depending on a condition, one part of the block will be executed and not the other one.

3.4.1. Sequence diagram of Use Case 1. Loading the model from compiled java classes

In this use case, the user loads the model represented in a collection of compiled java classes (.class files). The basic sequence is as follows:

- In the directory, load the .class files found. Repeat this process for each subdirectory found under the specified input directory.
- For each loaded class file, analyze its persistence configuration asking for each JPA annotation.
    - First, see if it's an entity and process its persistence configurations
        - Process the annotations associated with entities, and process their attributes, methods and associations.
    - See if it's an embeddable (value object) and process its persistence configurations
        - Process the annotations of the attributes and methods.
    - See if it's a mapped superclass and process its persistence configurations.
        - Process the annotations associated with mapped superclasses, and process their attributes and methods.
- Once all the classes have been analyzed and the intermediate model has been created, iterate over the different entities and their relationships to update them.
- The program returns the loaded domain model.

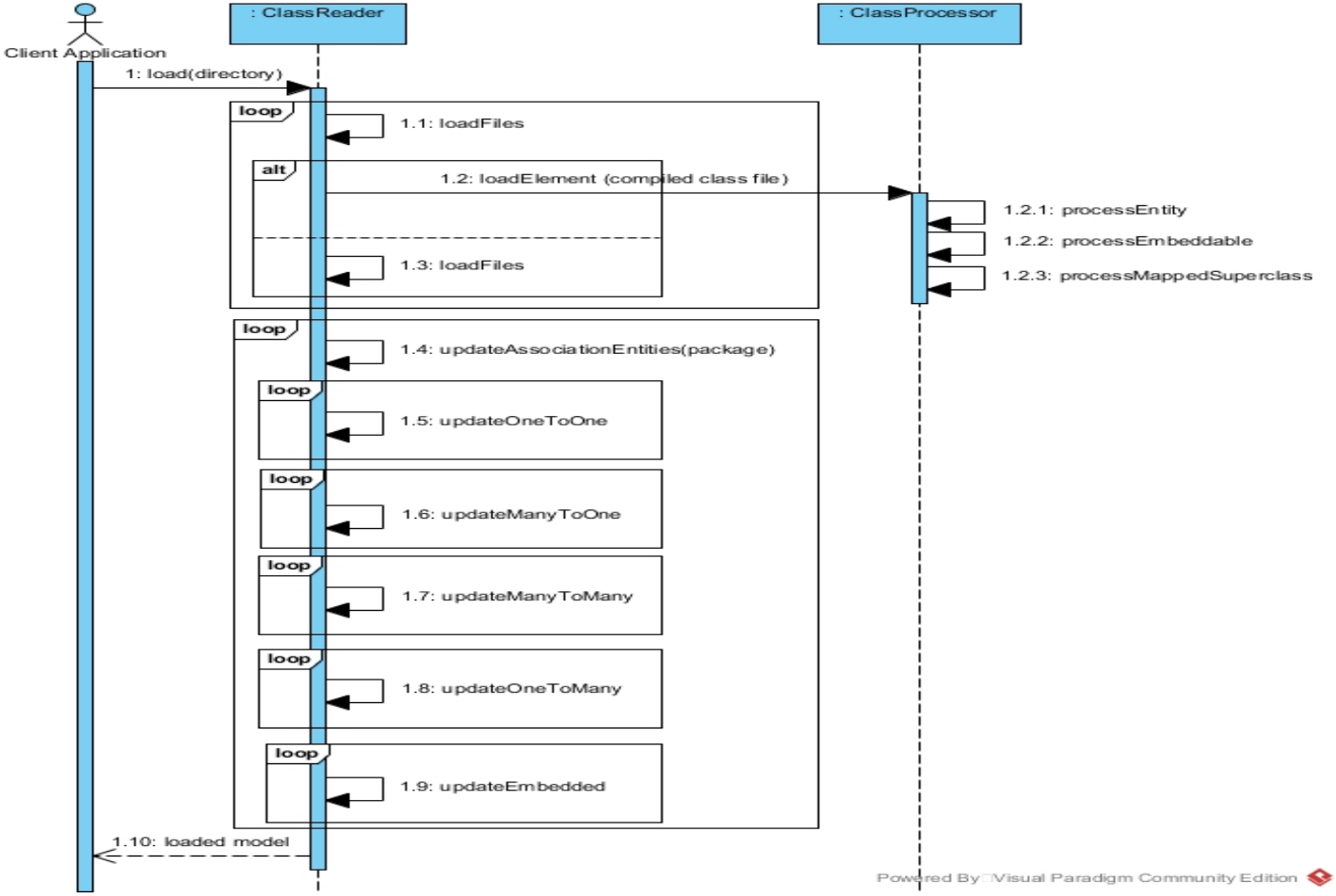

Figure 76. Sequence diagram: Loading the model from compiled java classes

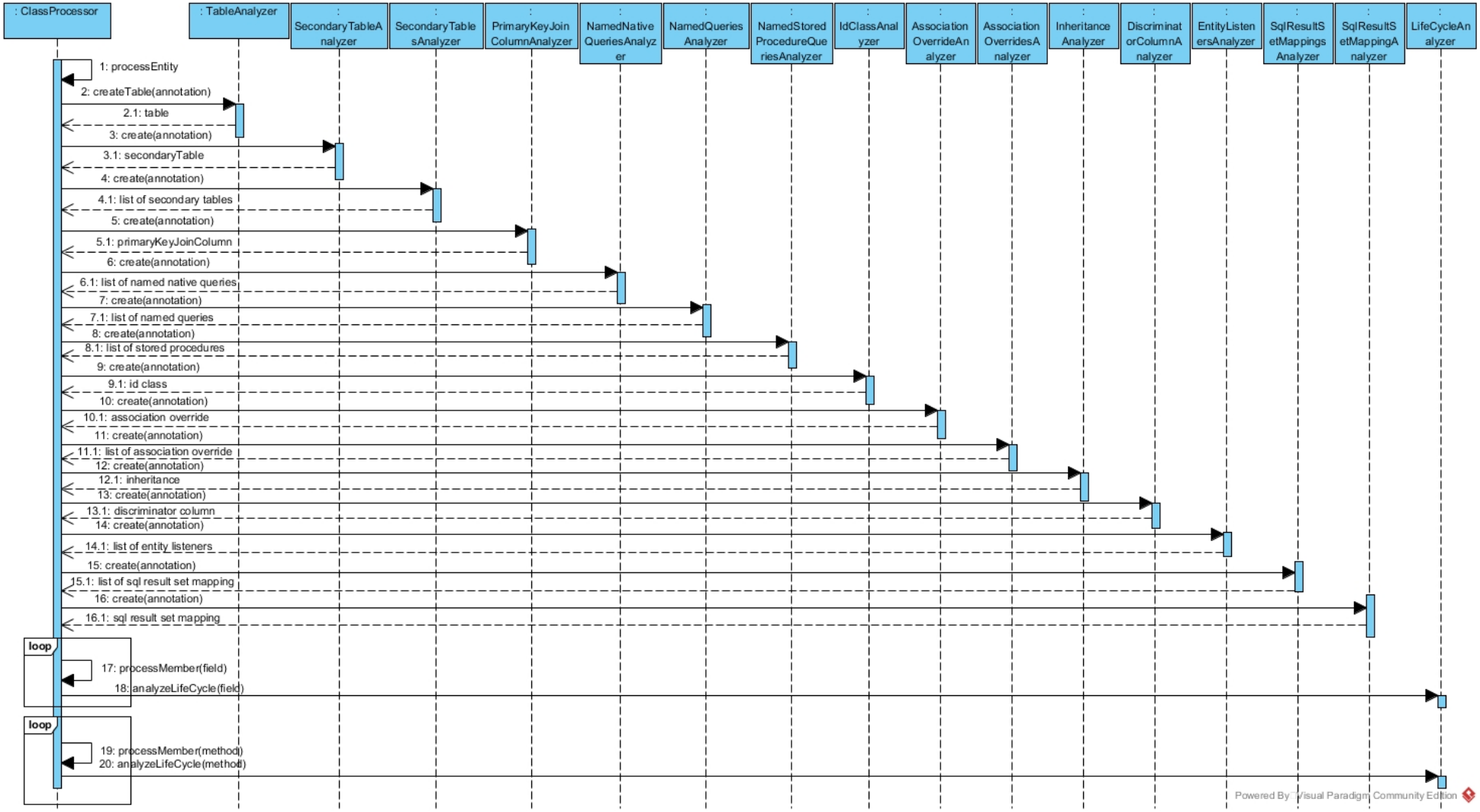

Figure 77. Sequence diagram: Load model from compiled classes: process entity

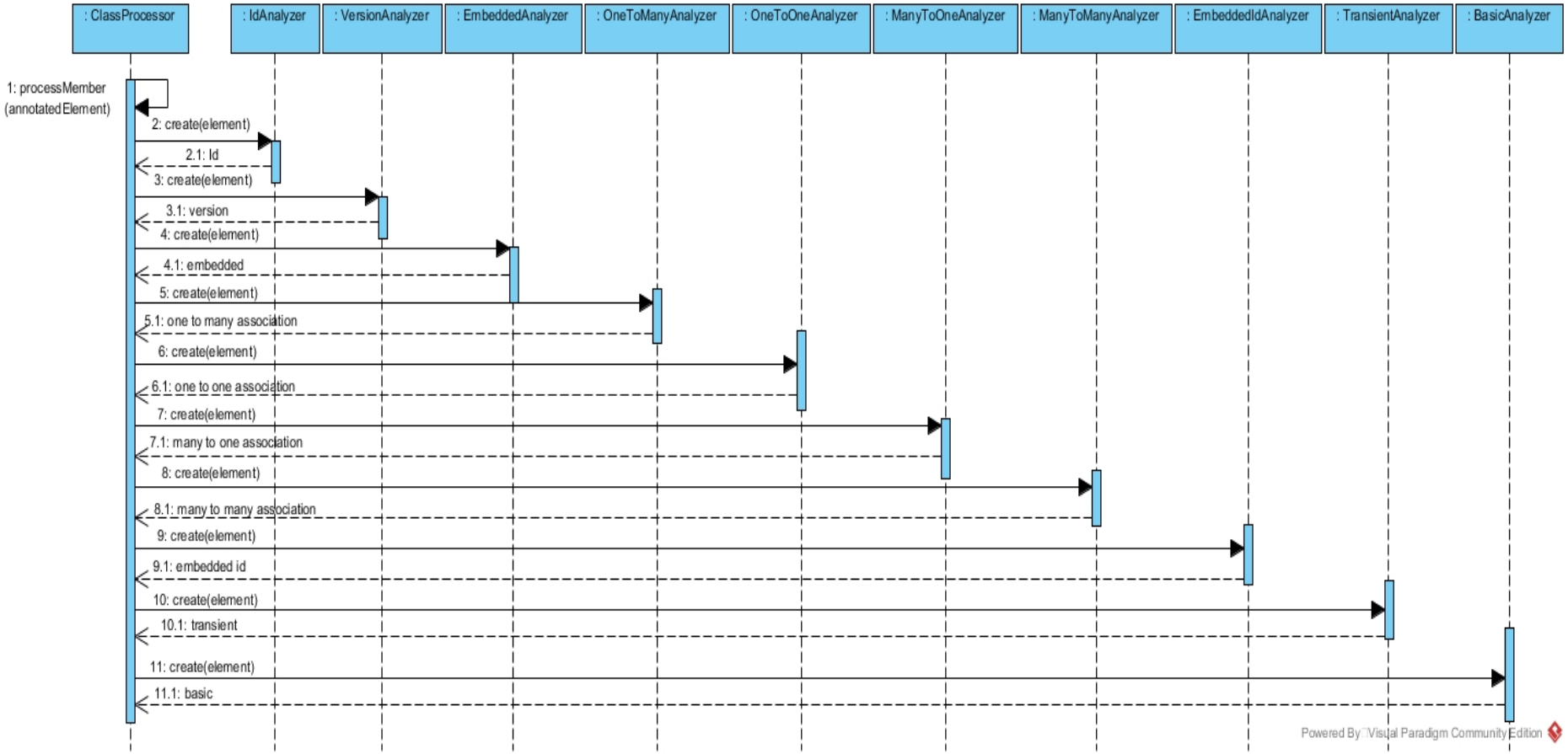

Figure 78. Sequence diagram: load model from compiled classes: process member

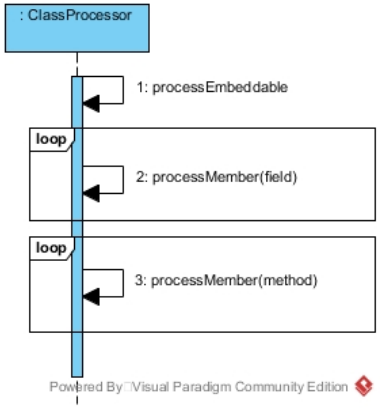

Figure 79. Sequence diagram: load model from compiled classes: process embeddable

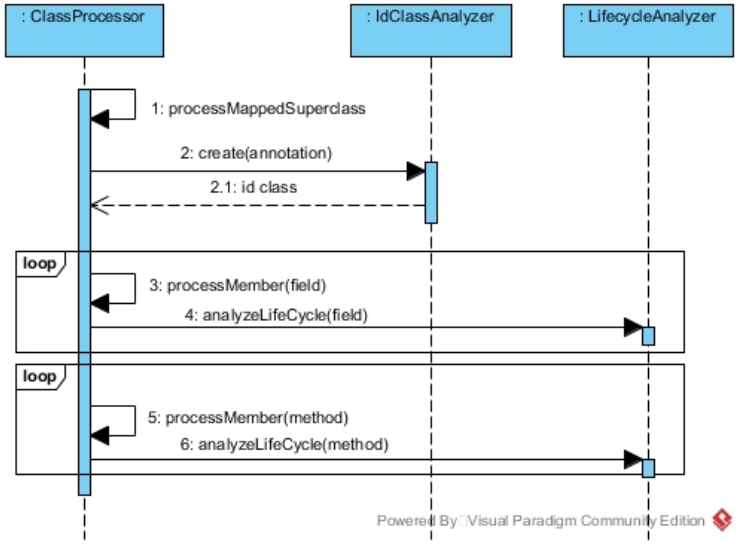

Figure 80. Sequence diagram: load model from compiled classes: process mapped superclass

3.4.2. Sequence diagram of Use Case 2. Loading the model from java source code

In this use case, the client loads the model expressed in java source code files (.java files). The basic process is as follows:

- The client specifies the directory where the java files are located.
- The program sets up the compiler in the user's machine to compile the java files, output the compiled class files in the directory and use the hibernate library to process persistence JPA annotations.
- The program searches for .java files and loads them. The searching process is repeated for each subdirectory under the specified input directory.
- Once set up, the program asks the compiler to compile the loaded java files.
- The program stores the original source code content for each of the loaded files.
- The program redirects the execution to the module that loads the domain model from compiled java files.
- When this second module returns the loaded model, this one redirects this loaded model and returns it to the client application.

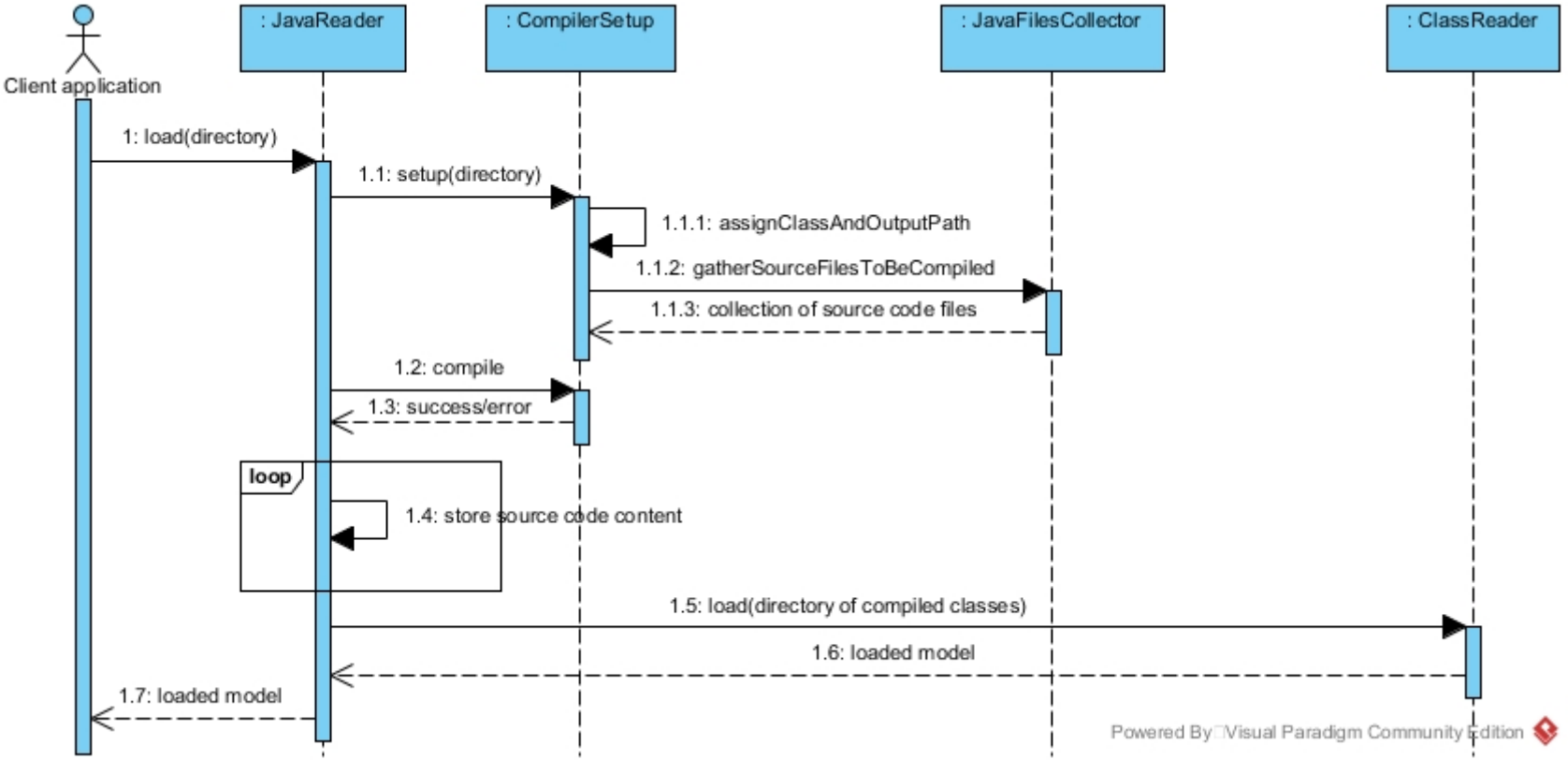

Figure 81. Sequence diagram: load model from java source code

3.4.3. Sequence diagram of Use Case 3. Loading the model from UML model files

In this use case, the client loads the model from UML model files. The basic process is as follows:

- The client specifies the location to find the .uml file and the program loads it.
- The program loads the root elements and the custom UML profile used to identify version attributes and value objects.
- Once the UML elements have been correctly loaded, the program iterates over them.
- If it's an entity, it creates an entity. If it contains the value object stereotype, it creates an embeddable.
- The program repeats the previous process for every package and subpackage loaded from the UML model file.
- When an entity is created, the UML elements of the element that represented the entity are analyzed to create the entity's inner elements.
- When an embeddable is created, the UML elements of the element that represented the embeddable are analyzed to create the embeddable's inner elements, and associations to the entity embedding the value object are created.
- The program returns the intermediate loaded domain model.

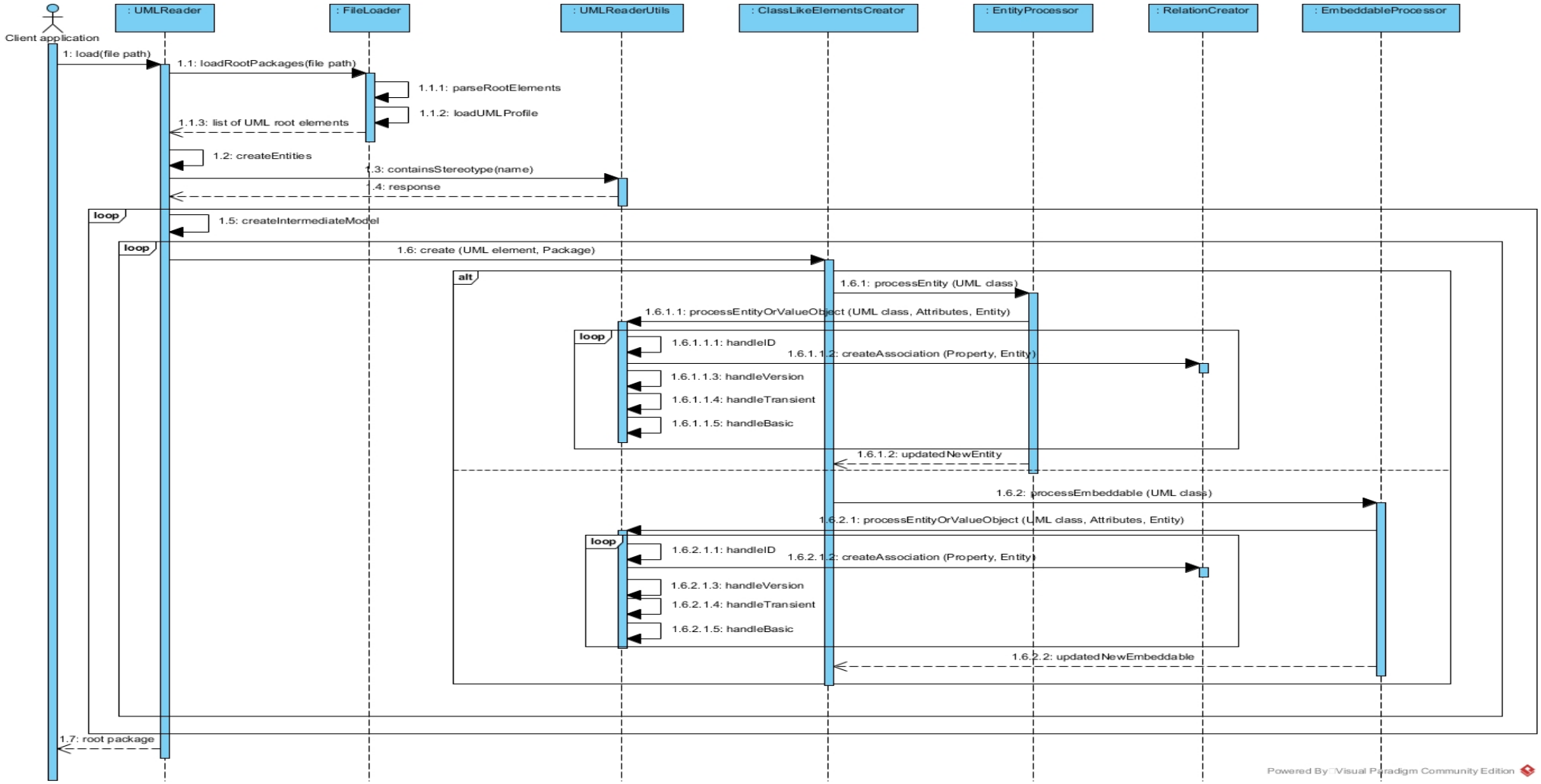

Figure 82. Sequence diagram of use case. Loading model from UML file

3.4.4. Sequence diagram of Use Case 4. Loading the model from a DSL file

In this use case, the client loads the domain model from a DSL file. The overall process is as follows:

- The client specifies the location of the .tfg file, and the program loads such file.
- The program compiles the file, obtains the semantic model and gets the contents.
- The program goes from the root element loaded to its children, iterating over them to create the different intermediate elements representing the loaded domain: entities, embeddables (value objects), or enumerations (custom types).
- Once the previous elements have been created, their inner elements and attributes are analyzed to complete them. This way, custom attributes specific of entities are analyzed and created, as well as operations and basic attributes. Similarly, basic attributes are analyzed and created for the value objects, or embeddables. Finally, the associations specified in the DSL file are analyzed, and registered in the intermediate loaded model.
- Once the intermediate model has been created, the program iterates over the entities loaded to see if a special element called *IdClass* must be created and updated.
- Finally, the program returns the loaded intermediate model.

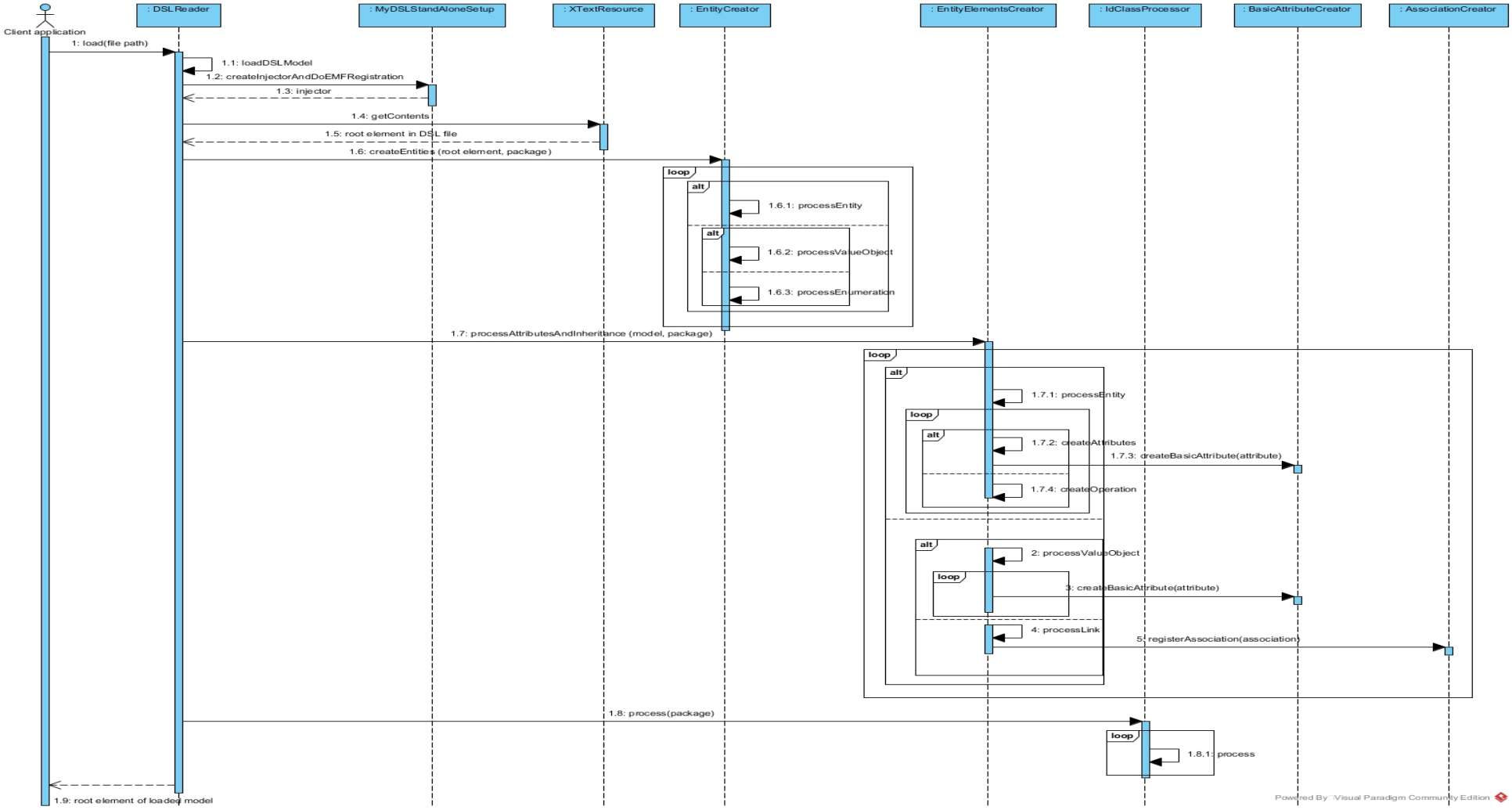

Figure 83. Sequence diagram of use case: load from DSL

3.4.5. Sequence diagram of Use Case 5. Loading the model from a database

In this use case, the client loads a domain model from a database. The basic process is as follows:

- The client specifies connection configurations and the program connects to the database.
- The program reads the metadata of the database, containing information about the database, its structure and elements.
- The program gets all the tables in the database from the metadata.
- For each table, it reads its identity information to create the identity of the entity, and checks if it's part of an association.
- It creates the entities from the information collected.
- For those elements that are part of the many to many associations, they are processed and these associations are created.
- For each entity created, the inner elements of their corresponding tables are read and analyzed, and the different entity attributes are created and assigned to the entities.
- The loaded model is then returned.

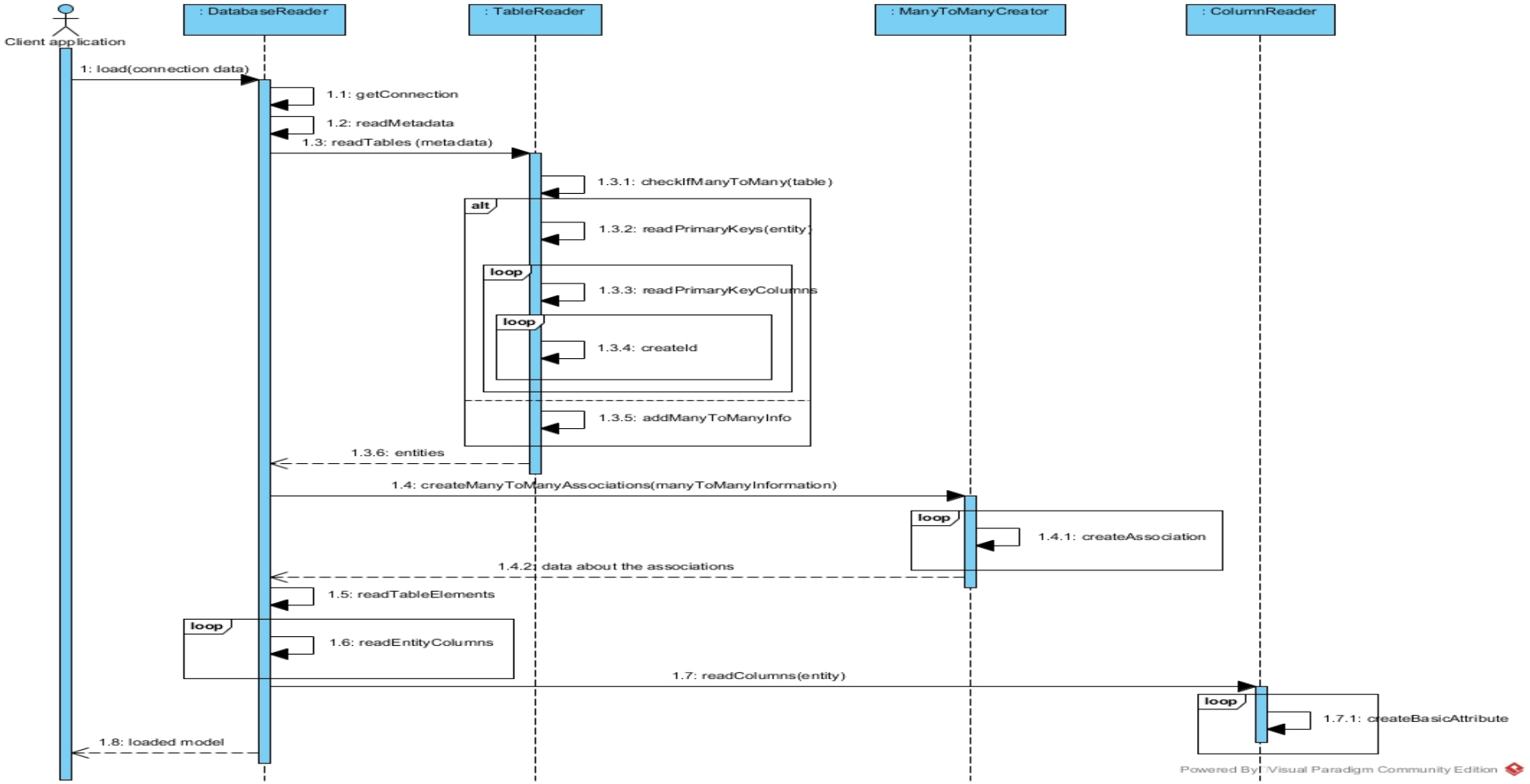

Figure 84. Sequence diagram: load model from database

3.4.6. Sequence diagram of Use Case 6. Express the domain essence with the custom DSL using its editor

In this use case, the DSL editor is configured and used by the end users to express the essence of the domain model. The overall process is as follows:

- If the user works with the plugin, it must be installed and integrated in his/her Eclipse workbench instance.
- Once the plugin has been integrated, or if the user is working with the Eclipse product, he/she must launch or start the Eclipse instance with the DSL editor feature integrated.
- The user creates a DSL file with the .tfg extension and the program will ask if he/she wants to apply the DSL editor. The user accepts.
- As the user writer the DSL file, autocomplete options are offered, as well as quick fixes and compile error detection.

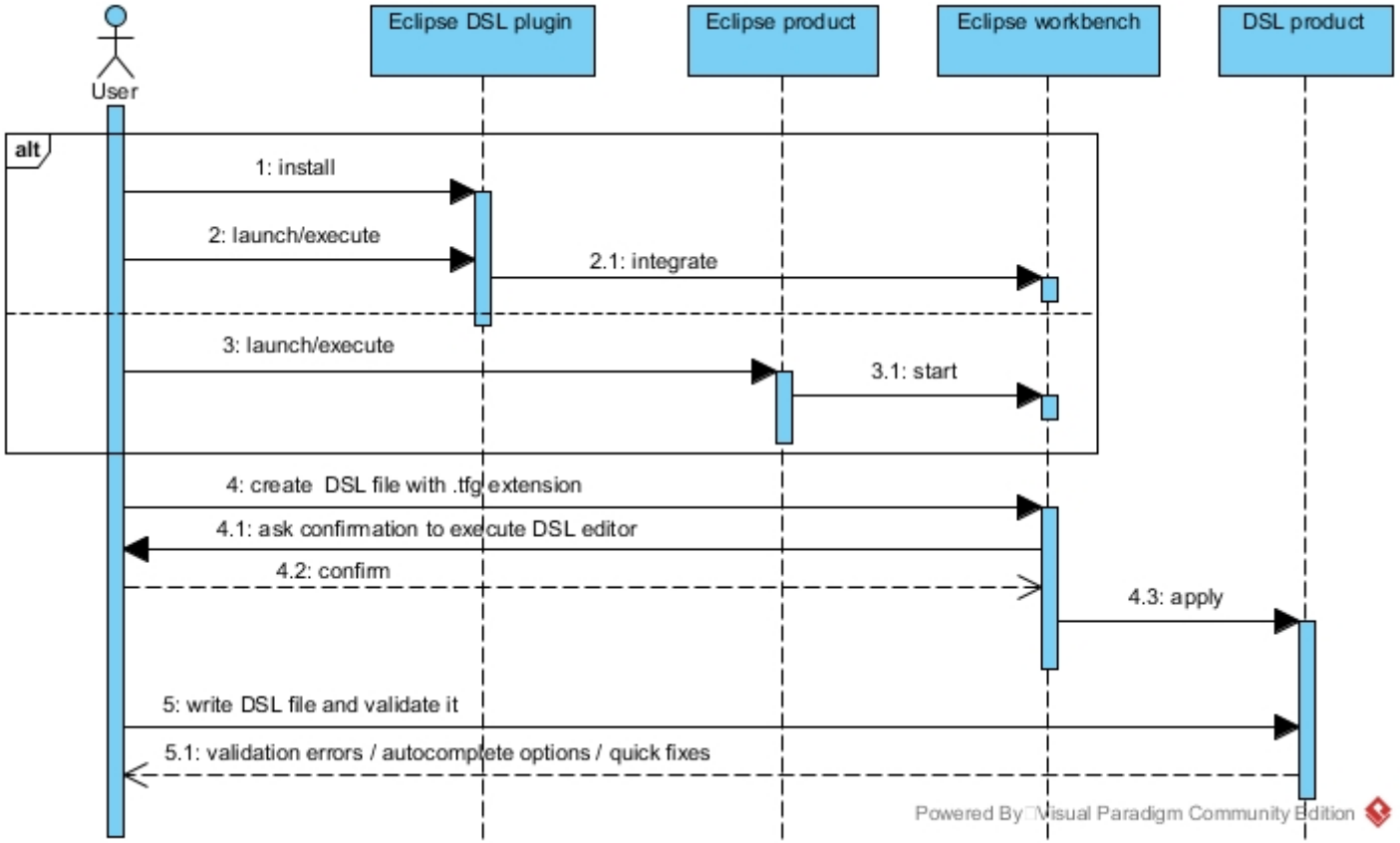

Figure 85. Sequence diagram: installing and using the DSL editor feature

3.4.7. Sequence diagram of Use Case 7. Generate the ORM and clean persistence annotations

In this use case, the client wants to generate the ORM persistence configuration file from the loaded model, and if source code content has been stored (the model has been loaded from java source code files), then this code is duplicated but with no persistence configurations in it. The overall process is as follows:

- The client provides the intermediate loaded model and specifies the directory where the conversion should take place.
- The program obtains the corresponding JAXB elements such as the marshaller, and creates the output directory if it doesn't exist.
- The program takes the intermediate domain model and transforms it into a format appropriate to data storage, such as .xml files, using the marshaller. The ORM file is created in the directory.
- If the domain contains original source code, it creates a new directory to store the new source code.
- The program loops or iterates over the loaded domain elements, and for each element if it has source code attached, it creates a new java source file and removes the persistence configurations in it.

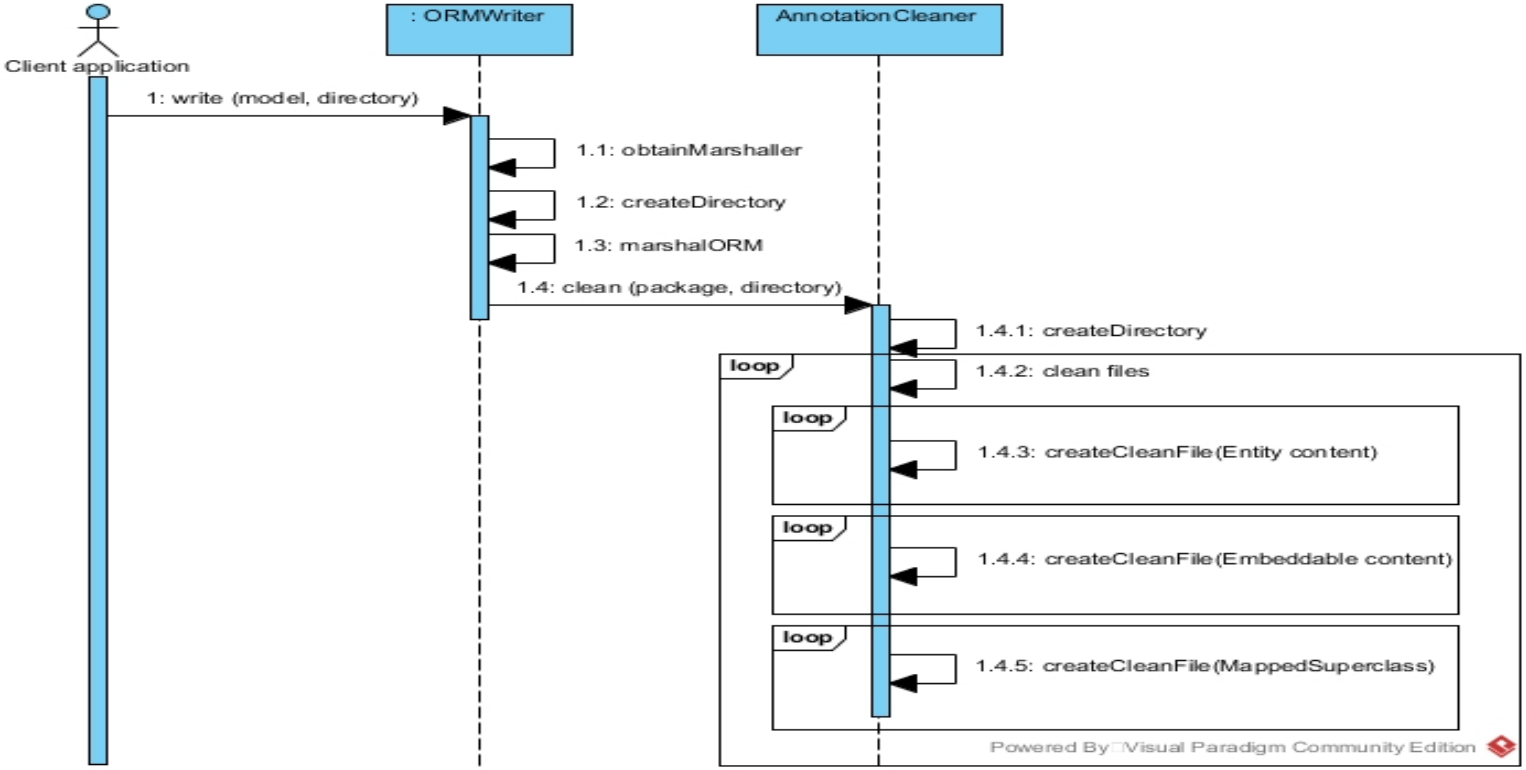

Figure 86. Sequence diagram of use case: Generate ORM file

3.4.8. Sequence diagram of Use Case 8. Generate Java Code

In this use case, the client wants to generate java source code files from the loaded domain model. The overall process is as follows:

- The client provides the loaded domain model and specifies the directory where the conversion should take place.
- The program creates such directory if it does not exist.
- The program iterates over the packages in the domain model, and in each package it iterates over the different elements representing the model (entities, value objects, custom types and mapped superclasses), creating the corresponding java classes, but only the skeleton of the classes.
- Once the java skeleton has been created, the program iterates again over the previous elements and it creates the different inner attributes and operations to work with the model.
- When associations are created and attached to java classes, the program adds special methods to handle these associations to a special class in charge of managing these relationships.

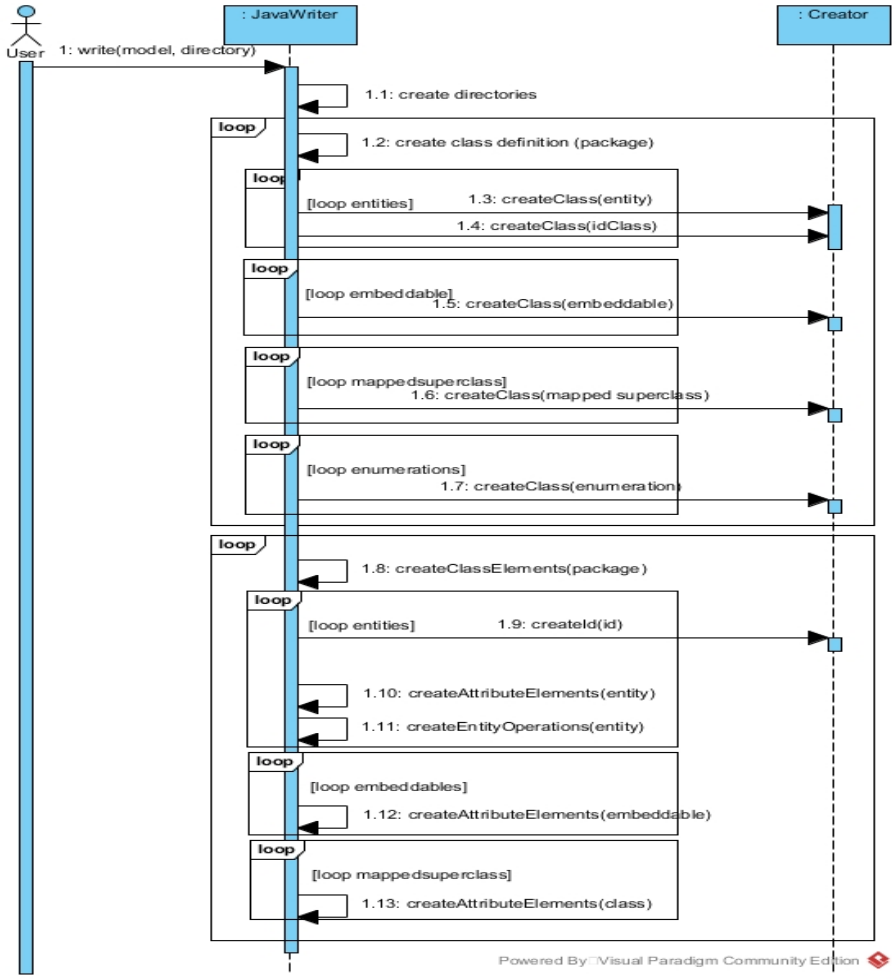

Figure 87. Sequence diagram: generate java model

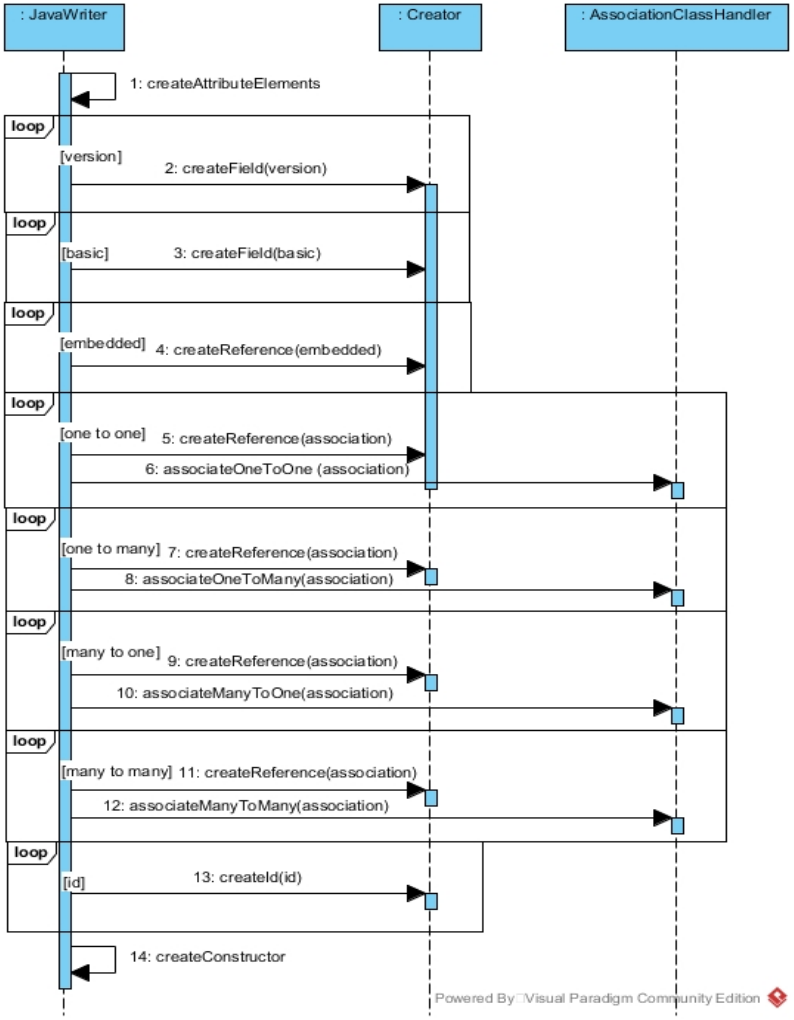

Figure 88. Sequence diagram: generate java model: create attribute elements

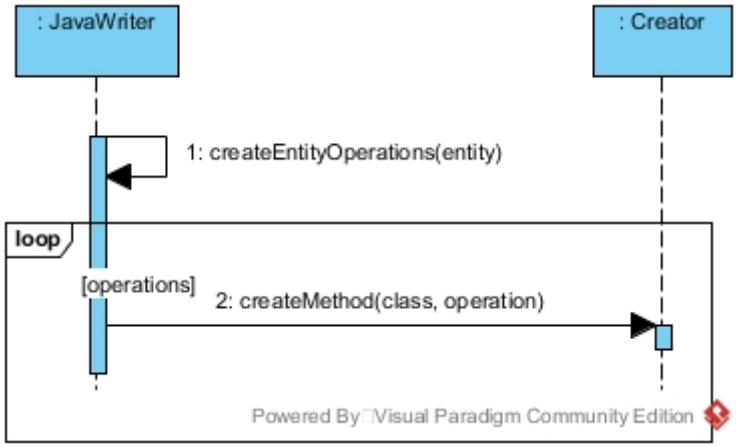

Figure 89. Sequence diagram: generate java model: create operations

3.4.9. Sequence diagram of Use Case 9. Generate a UML model file

In this use case, the client wants to generate a UML model file from the intermediate loaded domain model. The overall process is as follows:

- The client provides the intermediate loaded model and specifies the directory where the conversion should take place.
- The program creates such directory if it does not exist.
- The program creates the UML root element (a package), loads the custom UML profile used by the program and creates a duplicate in the output directory so that the client can have the .profile.uml file containing the UML profile and stereotypes applied in the UML model file to be created.
- The program creates the different basic primitive types used for methods and attributes.
- The program iterates over the packages in the loaded model, and it creates the UML package elements.
- In each package iteration, the program creates the different elements such as custom types (enumerations), value objects (embeddables) and entities.
- Once they have been created, the program iterates over them again and creates their inner elements and attributes.
- Finally, the program saves this UML model into the UML model file in the specified directory.

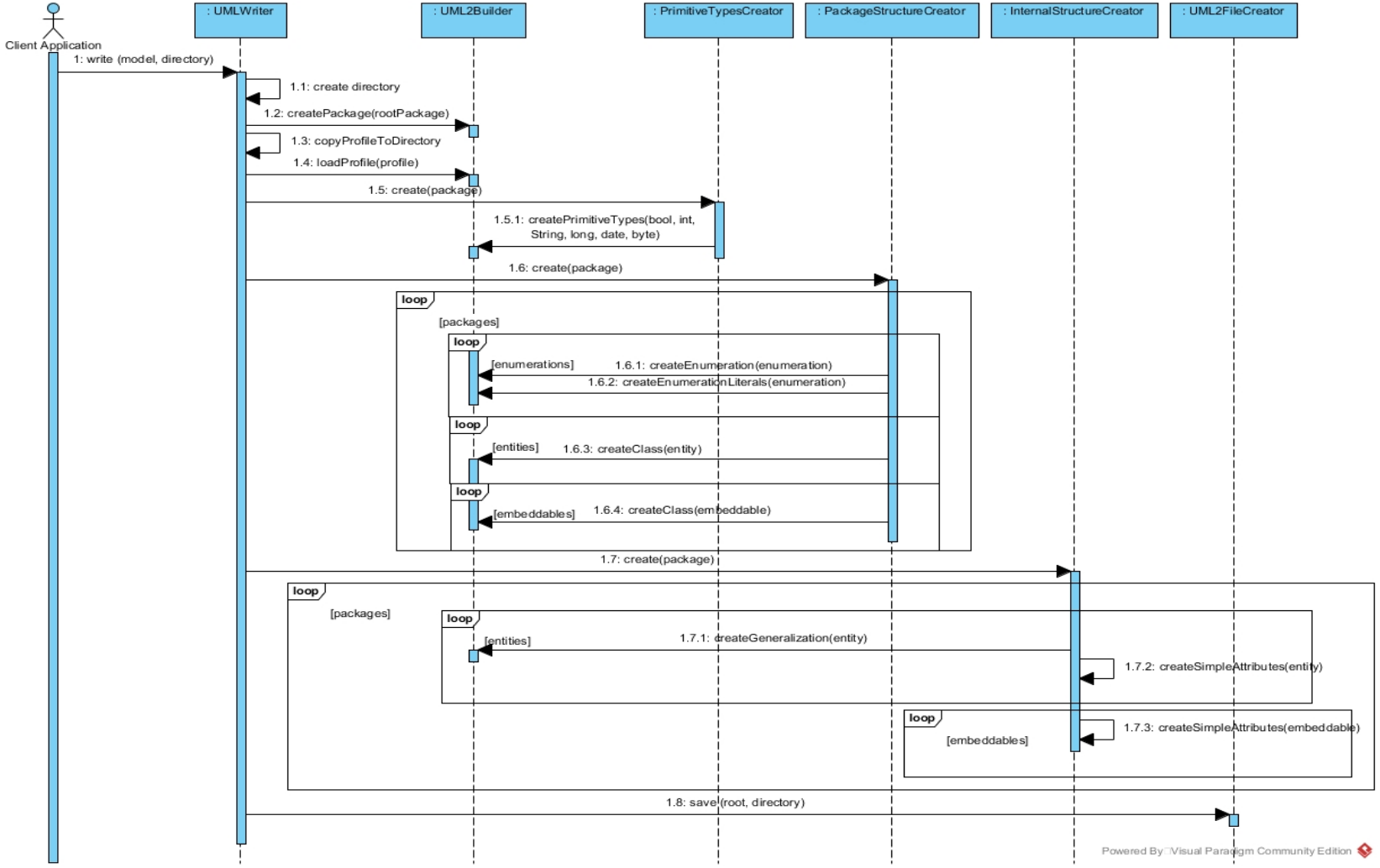

Figure 90. Sequence diagram: generate UML

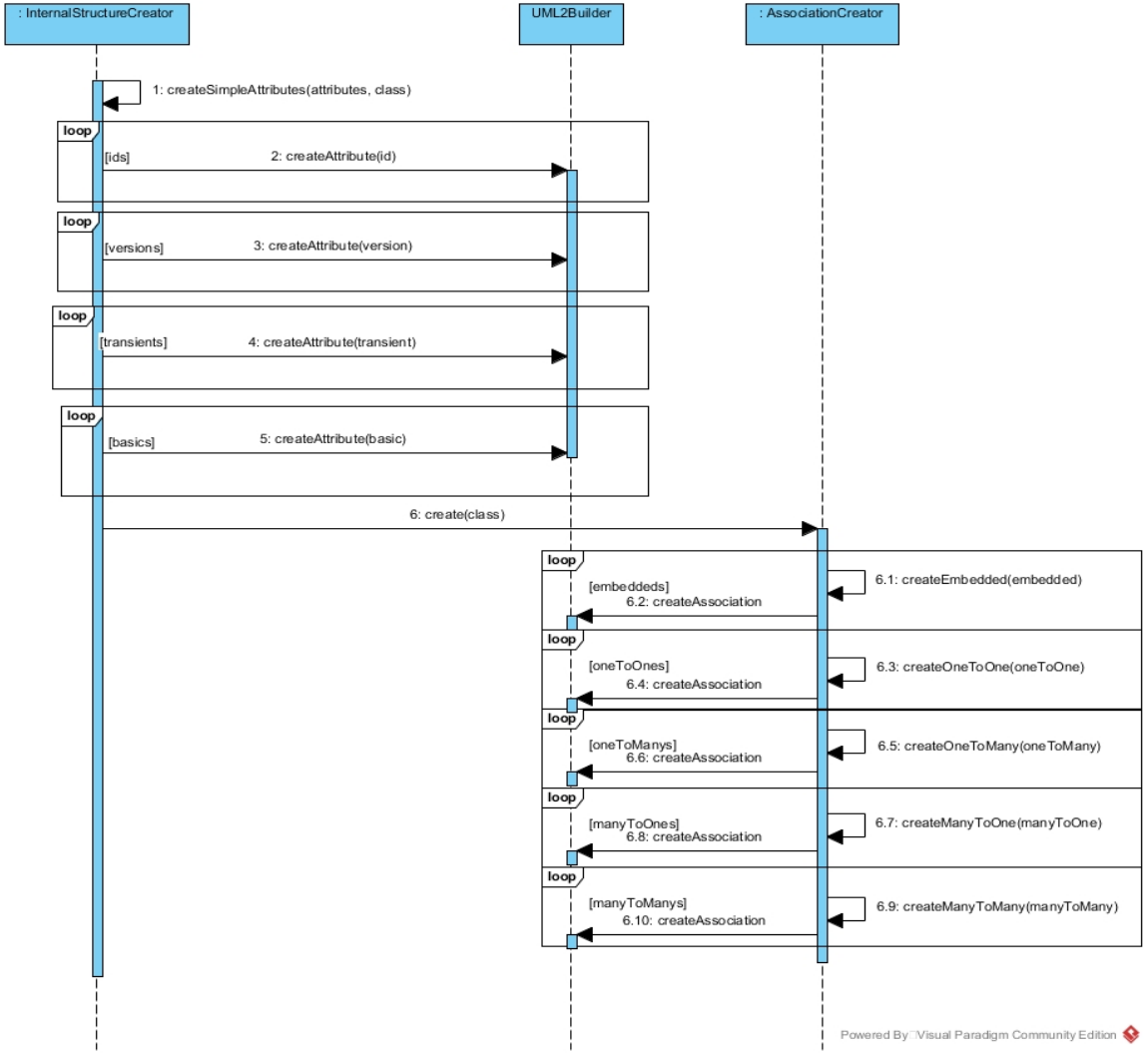

Figure 91. Sequence diagram: generate UML: generate simple attributes/references

3.4.10. Sequence diagram of Use Case 10. Generate database SQL scripts

In this last use case, the client wants to generate the database SQL scripts necessary to create the different tables and relationships to support and express the domain model in the database. The overall process is as follows:

- The client provides the loaded intermediate domain model and specifies the directory where the conversion should take place and the type of database.
- The program creates the directory if it does not exist.
- The program loads the corresponding database SQL template file corresponding to the specified database type. This template is used to script the different database elements.
- The program iterates over the different entities of the domain model, creating the corresponding SQL to generate a database table. Then, the program iterates over the attributes and associations of the entity, generating the SQL to create the inner elements and columns used to link tables. When an entity references a value object (embeddable), the program accesses and analyzes the embeddable, creating the SQL script of its contents and adding it to the entity associated with the embeddable.
- Once all the tables and their content have been scripted, the program creates the foreign key constraint used to link the different tables using the special columns created when associations were found.
- Finally, a .sql file is saved in the directory.

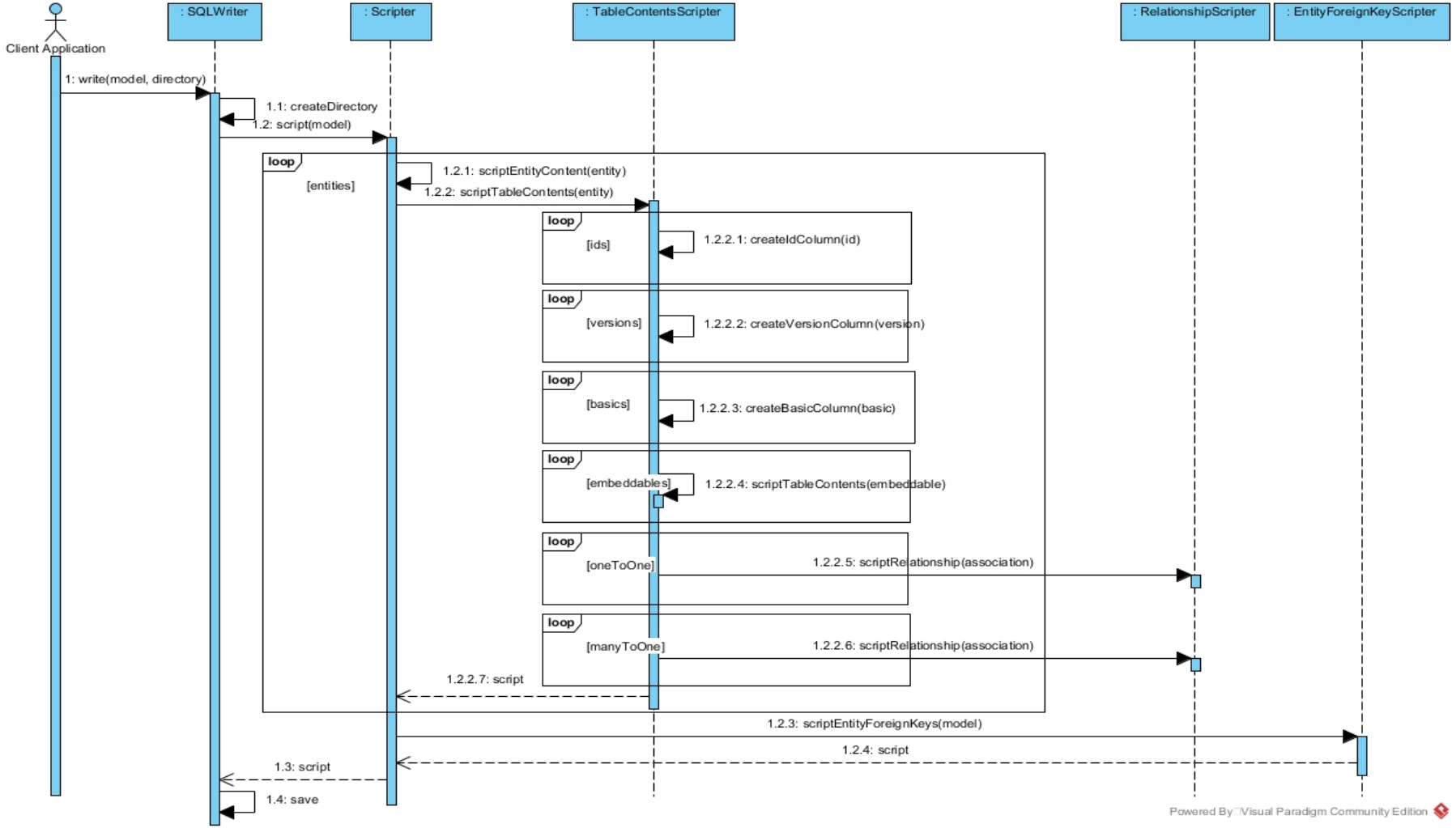

Figure 92. Sequence diagram: generate SQL.

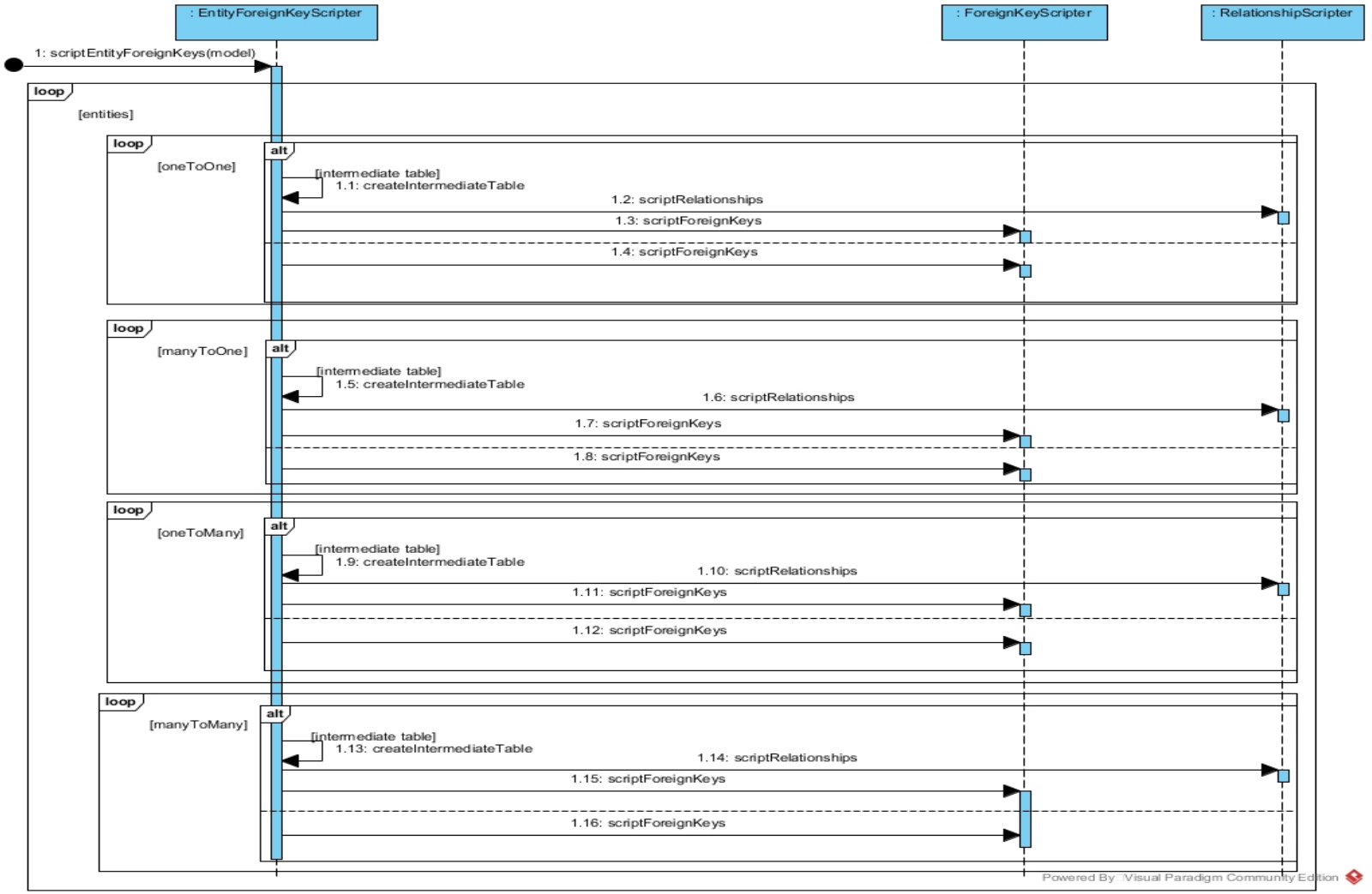

Figure 93. Sequence diagram: generate SQL: script foreign keys

4. Manuals

4.1. Command-line application manual

The command-line application is the product the user can execute directly from a terminal using a few basic commands to communicate with the application. While this product has some dependencies to a couple of different software libraries and the TFG core domain product, all dependency issues have been handled and the user does not need to worry about it, since all this is transparent.

The command-line application is downloaded as a runnable JAR file. A JAR file (Java ARchive), is a Java packaging format that can be imported as libraries in software projects, or that can be run in a console if the user has Java installed.

Therefore, once the user has the command-line application jar downloaded, he/she needs to have Java installed. In order for the command-line application to be able to interact successfully with the java compiler, the Java Development Kit needs to be installed and the jar needs to be executed using the JDK.

To install the Java Development Kit, the user should go to the Oracle's Download page[8], where the Java Platform (JDK) is available for free. Usually the Java compiler will get installed with the JDK, so in order to see if the installation was successful a new terminal should be opened and the user can check the version of the Java compiler to see if it's installed in the system by executing the following command:

```
javac –version
```

After that, everything should be fine and the command-line application should be ready to be executed.

To run the application, it is important to execute the java executable file from the JDK installed. To make sure this version is the one running, the user can go to the Environment Variables configuration page (in Windows, right click on *This PC* -> click the *Properties* option in the context menu -> Access the *Advanced System Settings* in the left menu panel -> Click the *Environment Variables* button at the bottom of the dialog). Then, check the *JAVA_HOME* variable is pointing to the installation directory of the JDK.

Then, open a terminal and launch the command-line application with the following command:

```
java –jar <path_of_the_cmd_jar_file>
```

An alternative to working with the previous environment variables is to work with the Java executable file in the JDK folder. To do this, execute the following command:

```
<path_java_executable_in_jdk> -jar <path_of_ cmd_jar_file>
```

The application will show the initial message explaining briefly its basic functionality, as shown in Figure 94.

[8] http://www.oracle.com/technetwork/java/javase/downloads/index.html

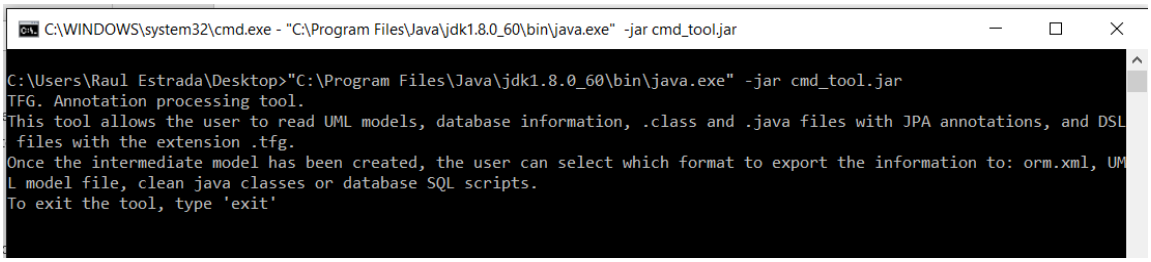

Figure 94. Command-line application: Welcome message

Then the user can start interacting with the application. A help message is offered to the user explaining the services and the commands to execute those services, as well as their structure and parameters. To show the help message, the user can write **`help`** at any moment.

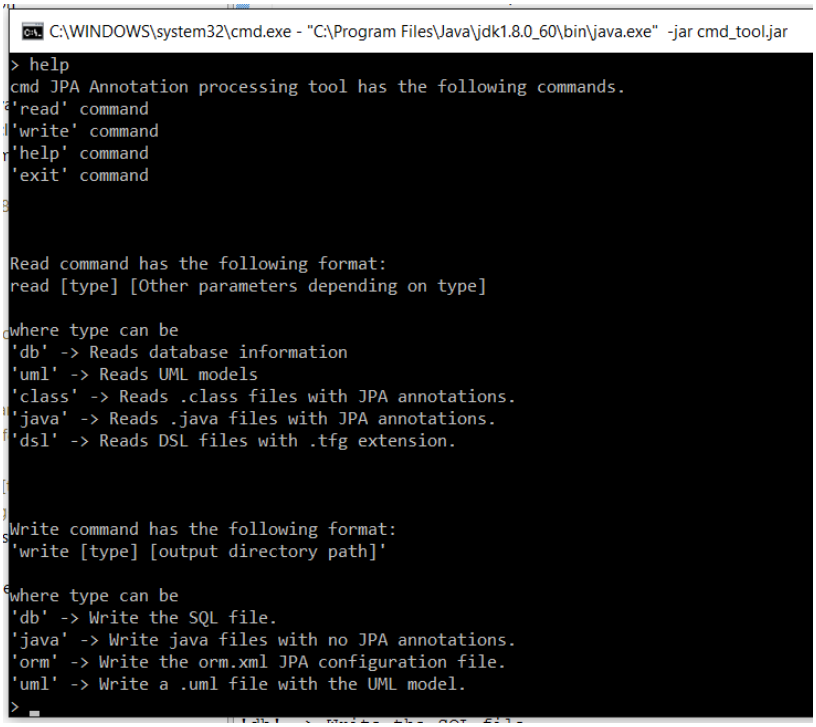

Figure 95. Command-line application: Help message

To quit the application, the user can type **`exit`** as shown in Figure 96, or press Ctrl + C.

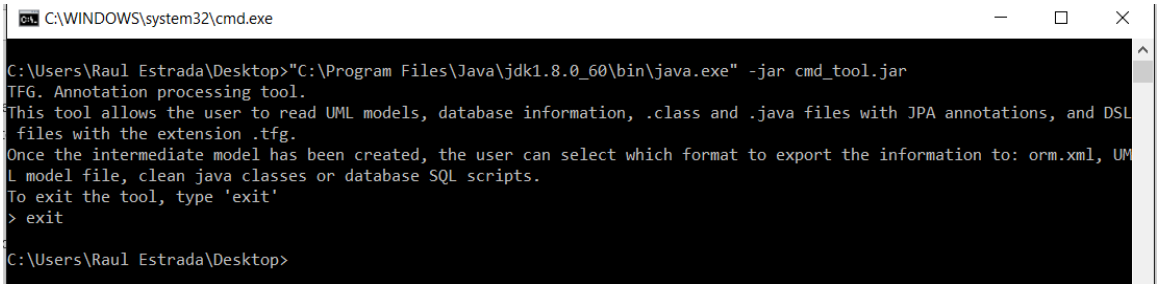

Figure 96. Command-line application: exit

There are two main services offered: load a given model, and perform conversions. Loading features are executed with the **`read`** command, while conversions are done with the

`write` command. The user can execute commands in any order, as any times as desired, but it should be noticed that a model must be loaded before any conversion can be made.

4.1.1. Read options

The tool can load the model from different sources, and therefore an additional parameter needs to be passed to the read command.

To load the model from compiled java files (.class files), the **`read class <directory>`** needs to be executed. The directory parameter is the path of the directory containing the compiled classes, and it should be noted that sub directories under the specified root directory will be scanned for files as well. The path should be between double quotation marks. Once the domain model has been loaded, a message informing about the number of elements loaded will be displayed through the screen.

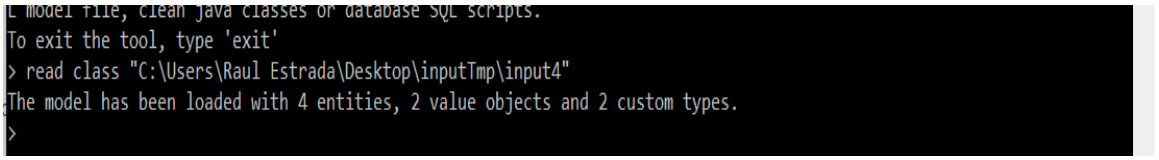

Figure 97. Command-line application: read class

A new model can be loaded again.

To load the model from java source code files (.java files), the **`read java <directory path>`** needs to be executed. The directory parameter is the path of the directory containing the java source code files, and all sub directories under the specified root directory will be scanned for files as well. The path should be between double quotation marks. Once the domain model has been loaded, a message informing about the number of elements loaded will be displayed through the screen.

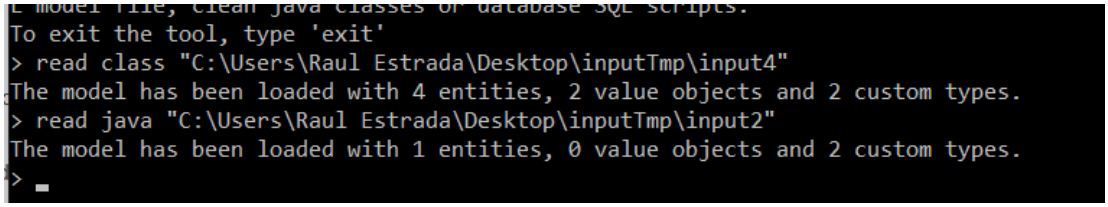

Figure 98. Command-line application: read java

To load the model from a UML model file (.uml file), the **`read uml <file path>`** needs to be executed. The file path parameter is the path of the UML model file containing the domain model. This path should be between the double quotation marks, and no UML profiles or stereotypes should be included, unless they are the ones established in the custom UML profile file that is created when a UML conversion is performed. After the domain model is successfully loaded, a message informing about the number of elements loaded will be displayed through the screen.

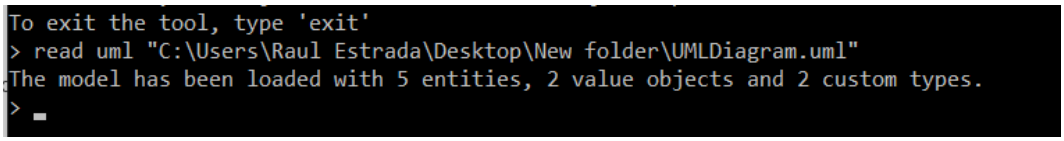

Figure 99. Command-line application: read UML model

To load the model from a custom DSL file (with the .tfg extension), the **`read dsl <file path>`** needs to be executed. The file path parameter is the path of the DSL file containing the domain model. This path should be between double quotation marks. While the tool is loading and compiling the DSL file using XText, XText messages are shown in the terminal with information about the loading and compiling process. After the domain model is loaded successfully, a message will be displayed informing about the number of elements loaded.

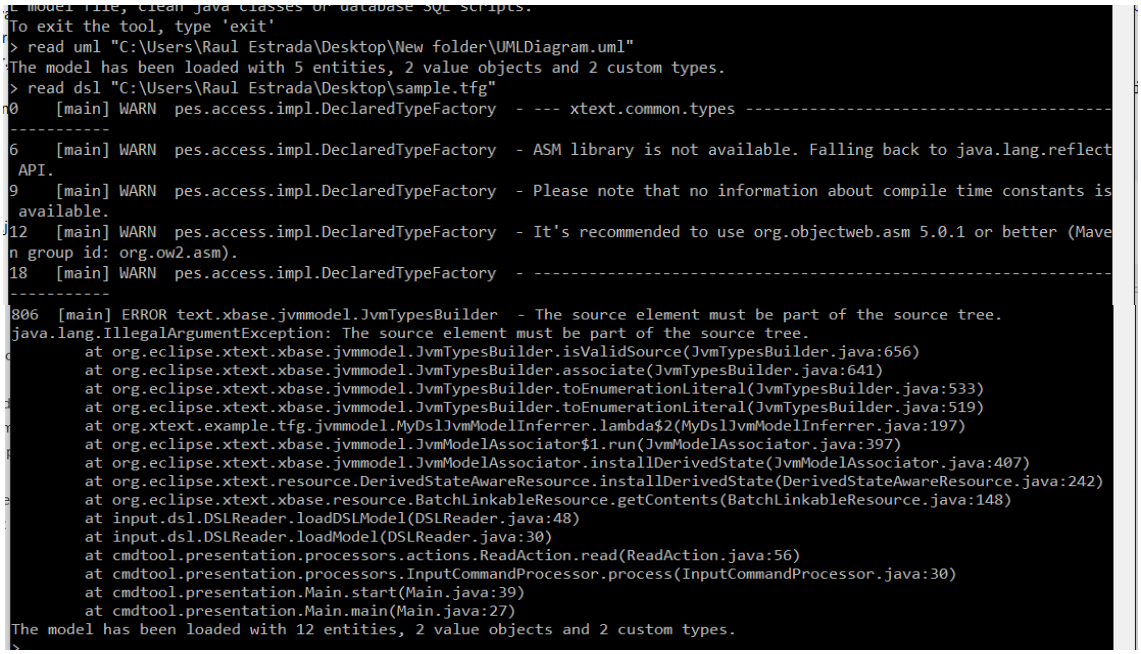

Figure 100. Command-line application: read dsl

To load the model from a database, some additional parameters are needed in the read command. Its structure is:

`read db type=<TYPE> host=<HOST> port=<PORT> name=<DATABASE_NAME> username=<USERNAME> password=<PASSWORD>`

The type, host, port, name, username and password parameters can be introduced in any order, but all of them are necessary and must be introduced. The values of the parameters are not enclosed with double quotation marks. The type parameter must be one of the following: ORACLE, MYSQL, POSTGRESQL, HSQLDB.

Finally, after the domain model is loaded successfully, a message will be displayed informing about the number of elements loaded.

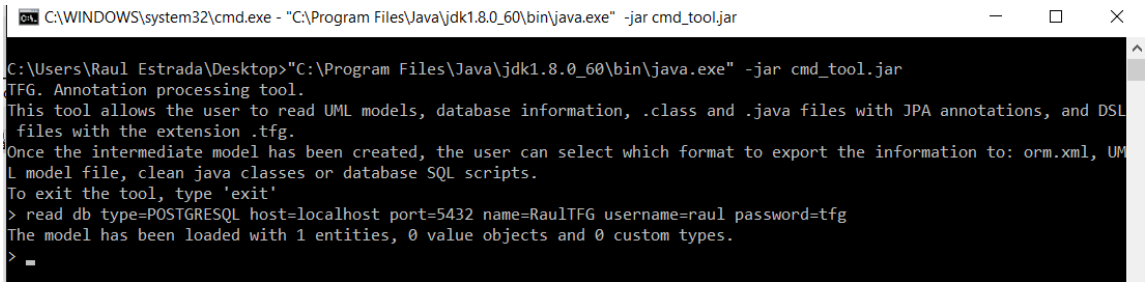

Figure 101. Command-line application: read from database

4.1.2. Write options

The command-line application can also perform several conversions to generate new forms of the domain model. Before performing any conversion, the tool will check if a domain model has been previously loaded. If it has, it will proceed to serve the request. Otherwise, it will display a message informing the user about the situation and it will not proceed with the write service.

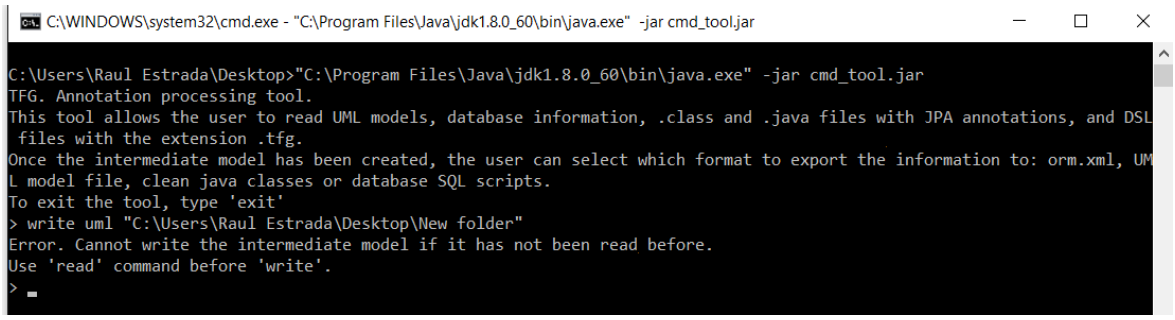

Figure 102. Command-line application: write before read

To generate the ORM persistence configuration file, the **`write orm <directory>`** command must be executed. The directory parameter is the path of the output directory, and must be enclosed in double quotation marks. Moreover, if the intermediate model has been loaded from java source code files, the original source code will be duplicated and persistence configurations will be removed from the new clean code.

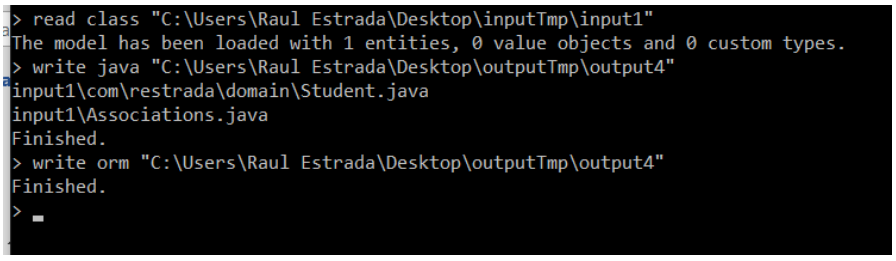

Figure 103. Command-line application: write orm

To generate a UML model file from the loaded model, the **`write uml <directory>`** command must be executed. The directory parameter is the path of the output directory, and must be enclosed in double quotation marks.

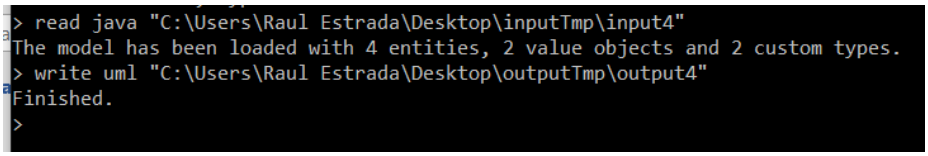

Figure 104. Command-line application: write uml

To generate the database SQL scripts from the loaded model, the **`write db <db type> <directory>`** command must be executed. The directory parameter is the path of the output directory, and must be enclosed in double quotation marks. The db type parameter is the type of database, and must be one of the following: ORACLE, MYSQL, HSQLDB, POSTGRESQL.

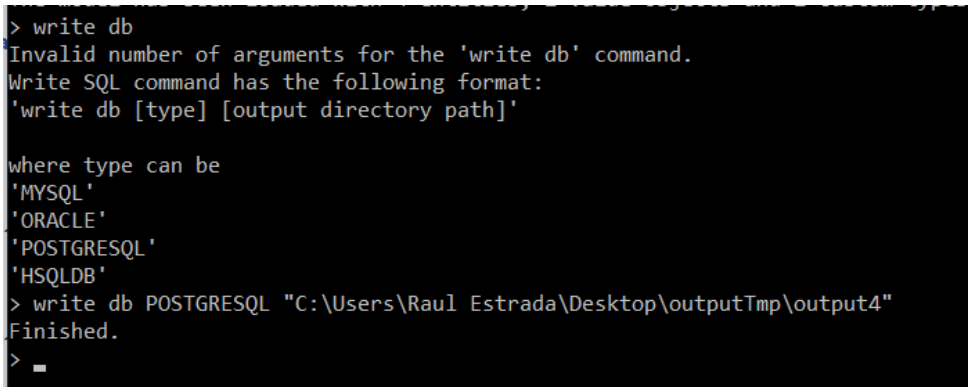

Figure 105. Command-line application: write db

To generate Java source code files from the loaded model, the **`write java <directory>`** command must be executed. The directory parameter is the path of the output directory, and must be enclosed in double quotation marks.

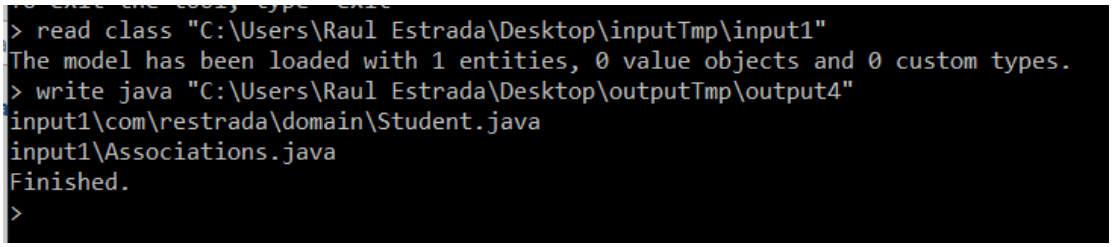

Figure 106. Command-line application: write java

4.2. Eclipse domain plugin manual

The Eclipse domain plugin can be downloaded as a zip file, which must be uncompressed. The new uncompressed folder will contain some configuration and project files, as well as come Java jars and two folders: *features* and *plugins*.

In the Eclipse instance of the user, he/she needs to go the *Help* menu in the menu bar, then access the *Install New Software…* menu item. A new window will be displayed. Click the *Add…* button.

A new window will open, the user must click the *Local…* button, select the new uncompressed file with the Eclipse domain plugin and accept. Back in the *Install* window, the new available software will be displayed and ready to be installed.

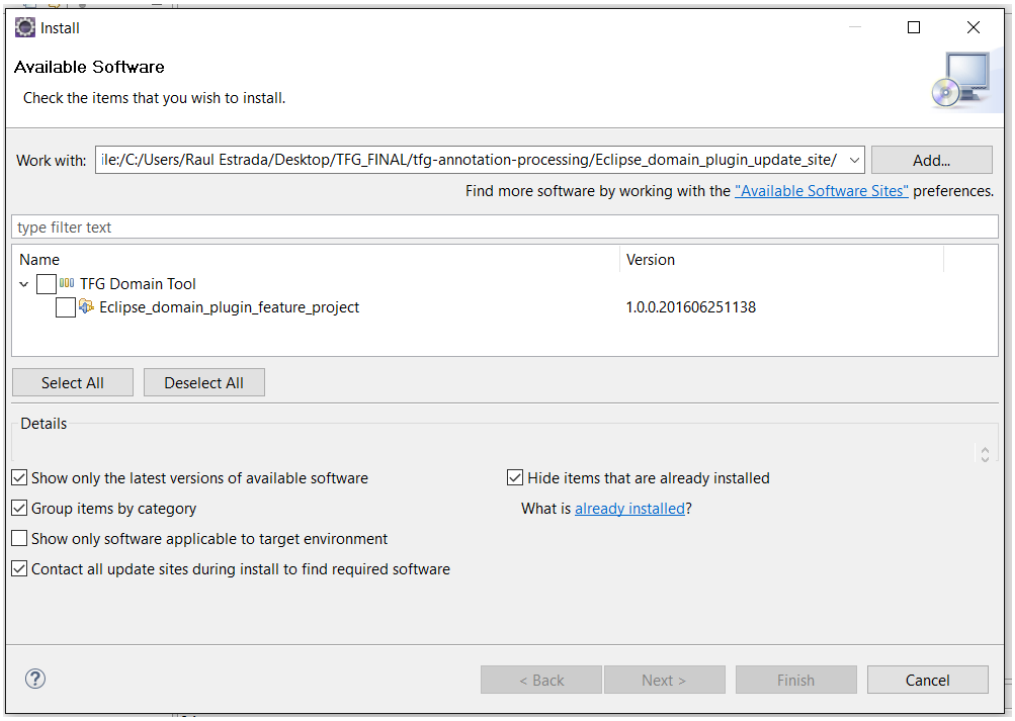

Figure 107. Eclipse Domain plugin. Installation

Select the feature and continue with the installation, accepting the different terms and conditions presented.

Once the plugin has been installed and integrated with the Eclipse instance of the developer, the tool provides two main options: load a model and perform conversions to generate a new form of the domain model. The conversion options will be disabled until a domain model has been loaded, and then the user can perform as many conversions as desired. Moreover, multiple loading operations can be performed as well.

The plugin offers its different services through popup menus, which are customized so that the correct loading option is displayed when the corresponding element is clicked and the menu appears.

4.2.1. Loading a model

To load a domain model from a collection of compiled java classes (.class), the user must click on the folder (folder, not package) containing such files. A context popup menu will appear, and a new menu item called *Annotation Processing Tool* will appear. This sub menu will contain the option to load the domain model from class files, as shown in Figure 108.

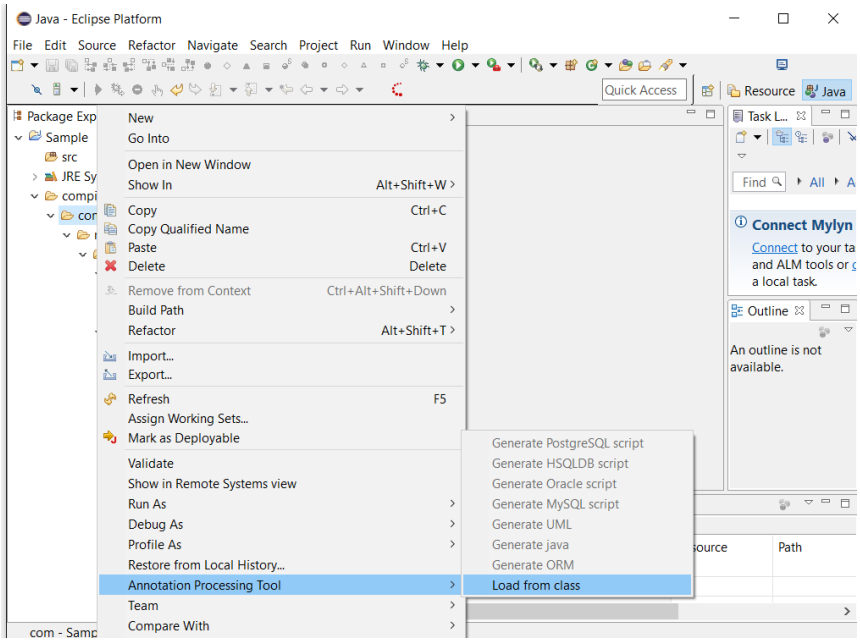

Figure 108. Eclipse domain tool: load classes

When the loading process finishes, a message will inform the user about the result of the operation.

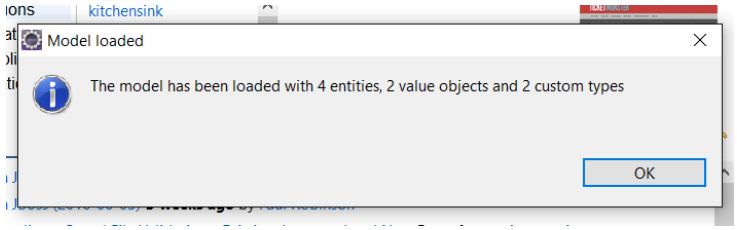

Figure 109. Eclipse domain tool: operation result message

To load the domain model from java source code files (.java files), the user must click on the java package. A context popup menu will appear, and in the *Annotation Processing Tool* submenu, the *Load from Java* option must be selected.

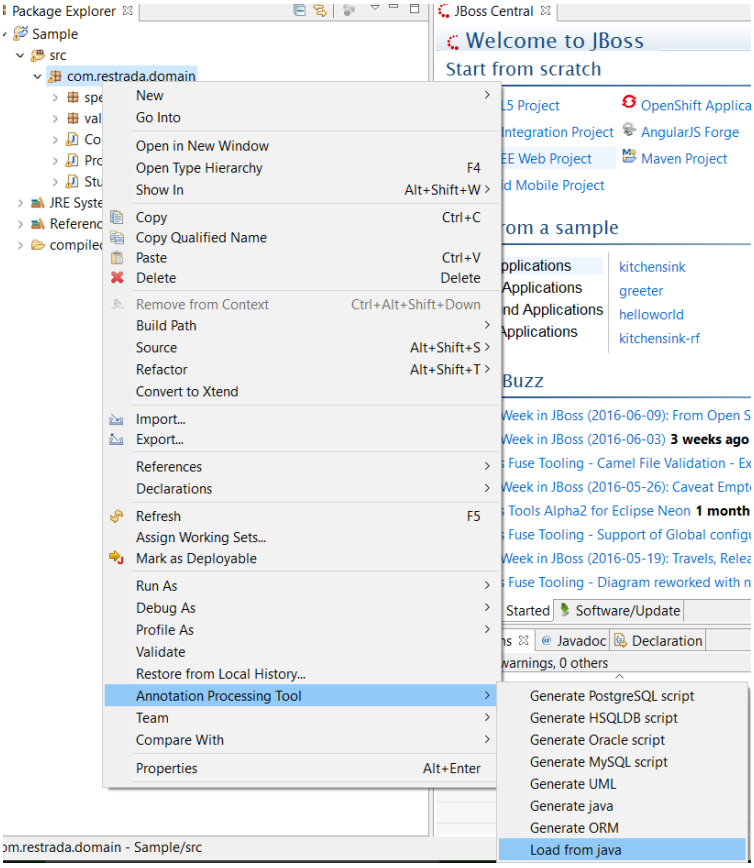

Figure 110. Eclipse domain tool: Load from java

Once the loading process is done, a message with the result of the operation will be displayed, as shown in Figure 109.

To load the domain model from a UML model file, the user must click on the .uml file. In the context popup menu, access the *Annotation Processing Tool* submenu, and select the *Load from UML* option.

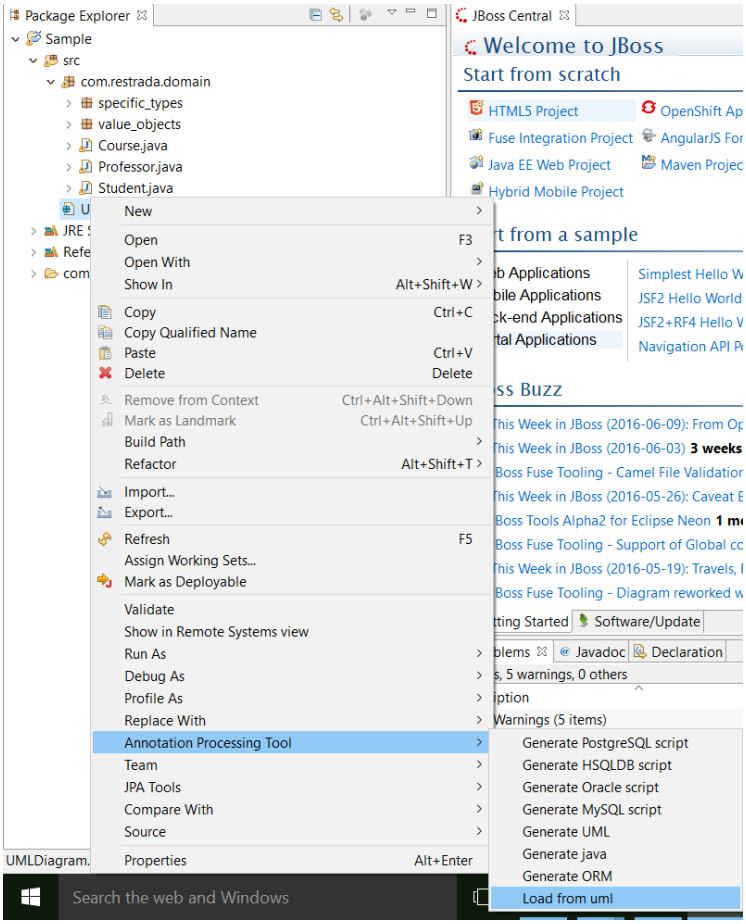

Figure 111. Eclipse domain plugin: Load from UML

Once the loading process is done, a message with the result of the operation will be displayed, as shown in Figure 109.

To load the domain model from a .tfg DSL file, the user must click on the .tfg file, and in the context popup menu, access the *Annotation Processing Tool* and select the *Load from DSL* option.

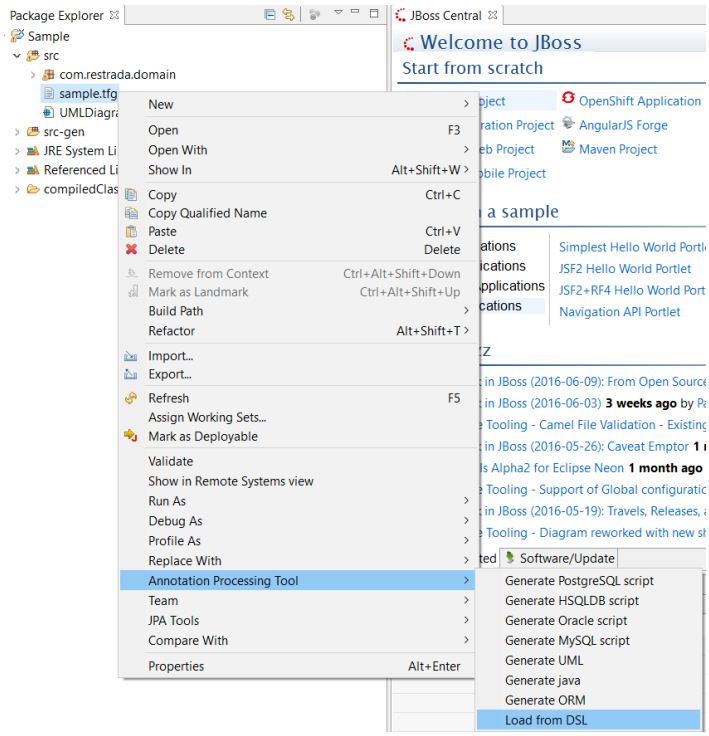

Figure 112. Eclipse domain plugin: Load from DSL

Once the loading process is done, a message with the result of the operation will be displayed, as shown in Figure 109.

To load the domain model from a database, the user must click on the Java project, and in the context popup menu, access the *Annotation Processing Tool* submenu and select the *Load from database* option. In the new dialog window, introduce all the configuration parameters needed to connect to the database.

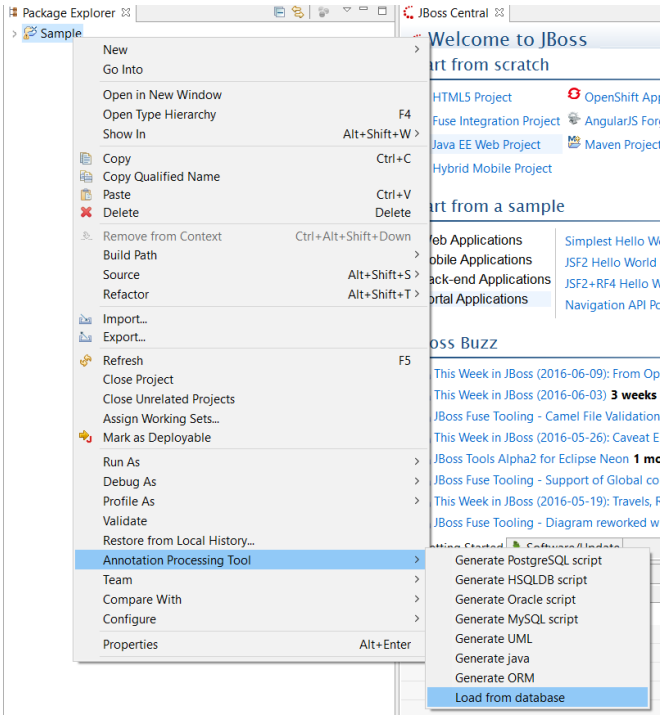

Figure 113. Eclipse domain plugin: Load from database

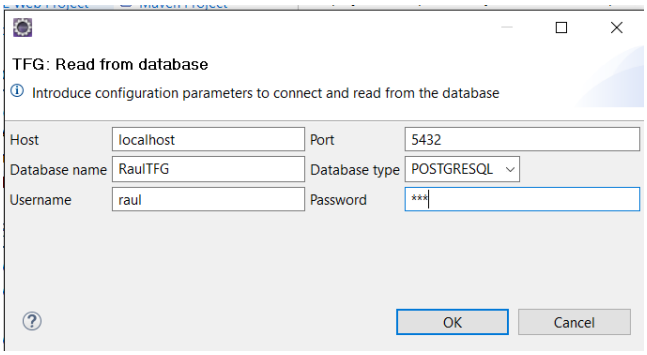

Figure 114. Eclipse domain plugin: database configuration

Once the loading process is done, a message with the result of the operation will be displayed, as shown in Figure 109.

4.2.2. Performing conversions

Once a domain model has been loaded, the different conversion options will become available. These options are available in each of the Eclipse plugin customized popup menu. That is, every popup menu that offered the possibility of loading a domain model offers also the possibility to perform any of the available and supported conversions.

In the different images displayed in Loading a model, page 142, the user can also see the different conversion options.

To generate the ORM configuration file from the loaded model (and a copy of the original java source code content without persistence configurations if the model was loaded

from .java files), the user must select the *Generate ORM* option in the *Annotation Processing Tool* submenu.

To generate java source code files from the loaded model, the user must select the *Generate java* option in the *Annotation Processing Tool* submenu.

To generate a UML model file from the loaded domain model, the user must select the *Generate UML* option in the *Annotation Processing Tool* submenu.

To generate the SQL database script from the loaded domain model, the user must select the *Generate <target database type> script* option in the *Annotation Processing Tool* submenu. The target database type is the type of the database where the script will be executed (MySQL, Oracle, HSQL or PostgreSQL).

The user can execute as many conversions as desired. After selecting any of the previous options to perform the different conversions, a new dialog window will appear so that the user can select the folder where the output of the conversion process must be placed.

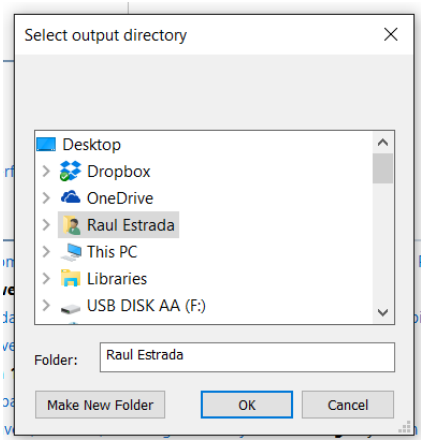

Figure 115. Eclipse domain plugin: select output folder

Once the conversion process is done, a message will be displayed to inform the user about the result of the operation.

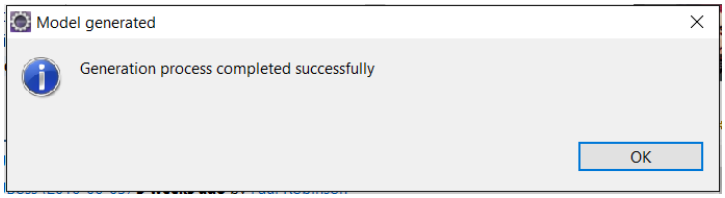

Figure 116. Eclipse domain plugin: output message

4.3. Maven plugin manual

The user will need to have Maven installed in his/her computer. Its official webpage offers a tutorial[9] on how to install this program. To check if Maven was installed successfully, the user can open a command-line or terminal and execute the **`mvn --version command.`**

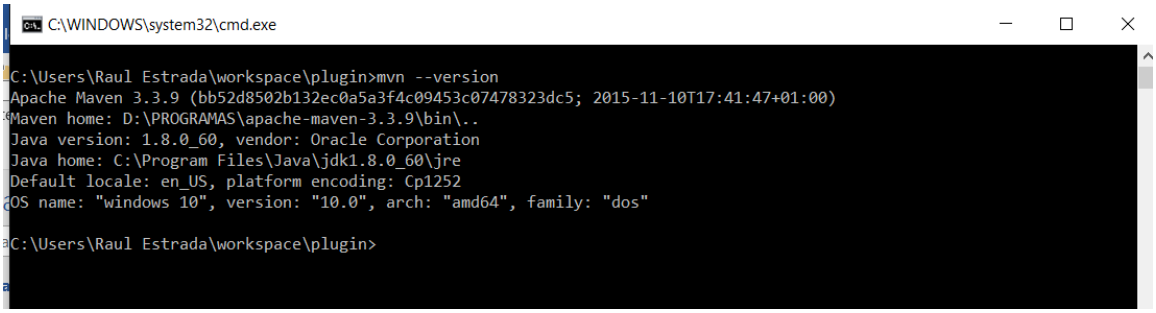

Figure 117. Check Maven installation

The Maven TFG plugin project can be downloaded compressed as a zip file. When uncompressed, the folder contains the Maven project. In the root directory, there's a pom.xml file with the configuration of the Maven plugin.

The tool can be configured to load a domain model and perform different conversions on that loaded model to generate new forms of the model. Only one model can be loaded, and multiple conversions are supported.

The domain model can be loaded from compiled java classes, java source code files, a UML model file, a .tfg DSL file and a database (PostgreSQL, Oracle, MySQL or HSQLDB). The user can configure the Maven plugin as follows:

- In the pom file, there's a `<toolConfig>` tag that contains all the configurations of the Maven plugin.
- A domain model must be loaded, and there are two different cases:
  - Load the model from a local directory or file: That is, the model will be loaded from compiled java classes, java source code files, a UML model file or a DSL file. In this case, the `<directoryInput>` tag is used.
    - Inside the `<directoryInput>` tag, the `<from>` tag specifies the path of the directory or file containing the domain model.
    - Inside the `<directoryInput>` tag, the `<type>` tag specifies the type of domain model to load (JAVA, CLASS, UML or DSL).
  - Load the model from a database. In this case, the `<databaseInput>` tag is used. Inside this tag, several other tags are used:
    - The `<host>` tag is used to specify the database host.
    - The `<port>` tag is used to specify the database port.
    - The `<name>` tag is used to specify the database name.

[9] https://maven.apache.org/install.html

- The <username> tag is used to specify the username to connect to the database.
- The <password> tag is used to specify the password to connect to the database.
- The <type> tag is used to specify the database type (MYSQL, POSTGRESQL, ORACLE, HSQLDB).

Once the loading operation has been configured, the output directory is next. After the <directoryInput> or <databaseInput> tag, a <toDirectory> tag is used to specify in which folder or directory the different conversions will take place. The absolute path of the directory must be placed here.

To indicate the conversions to be performed, the <conversions> tag is used. Inside this tag, the user can use the <param> element to specify as many conversions as desired. The values inside the <param> element can be TO_ORM_XML, TO_DATABASE_MYSQL_SCRIPT, TO_DATABASE_ORACLE_SCRIPT, TO_DATABASE_POSTGRESQL_SCRIPT, TO_DATABASE_HSQLDB_SCRIPT, TO_UML_MODEL and TO_JAVA.

```
</goals>
<configuration>
    <toolConfig>
        <directoryInput>
            <from>C:\Users\Raul Estrada\Desktop\NEW_TEST_APRIL\1\uml\UMLDiagram.uml</from>
            <type>UML</type>
        </directoryInput>
        <toDirectory>C:\Users\Raul Estrada\Desktop\NEW_TEST_APRIL\mvn\1</toDirectory>
        <conversions>
            <param>TO_ORM_XML</param>
            <param>TO_UML_MODEL</param>
            <param>TO_DATABASE_MYSQL_SCRIPT</param>
            <param>TO_DATABASE_ORACLE_SCRIPT</param>
            <param>TO_DATABASE_POSTGRESQL_SCRIPT</param>
            <param>TO_DATABASE_HSQLDB_SCRIPT</param>
            <param>TO_JAVA</param>
        </conversions>
    </toolConfig>
</configuration>
</execution>
```

Figure 118. Maven plugin: Load UML model file

Figure 121 shows an example of a Maven configuration to load the domain model from a UML model file, and perform all the possible conversions.

To execute the Maven plugin, the user must open a command-line or terminal in the uncompressed folder with the Maven plugin project, and execute the following command:

```
mvn clean
```

Then, to compile the plugin execute:

```
mvn compile
```

Package the plugin with:

```
mvn package
```

Then, execute:

```
mvn install
```

As the Maven plugin is executed, it will print information messages for testing and debugging purposes.

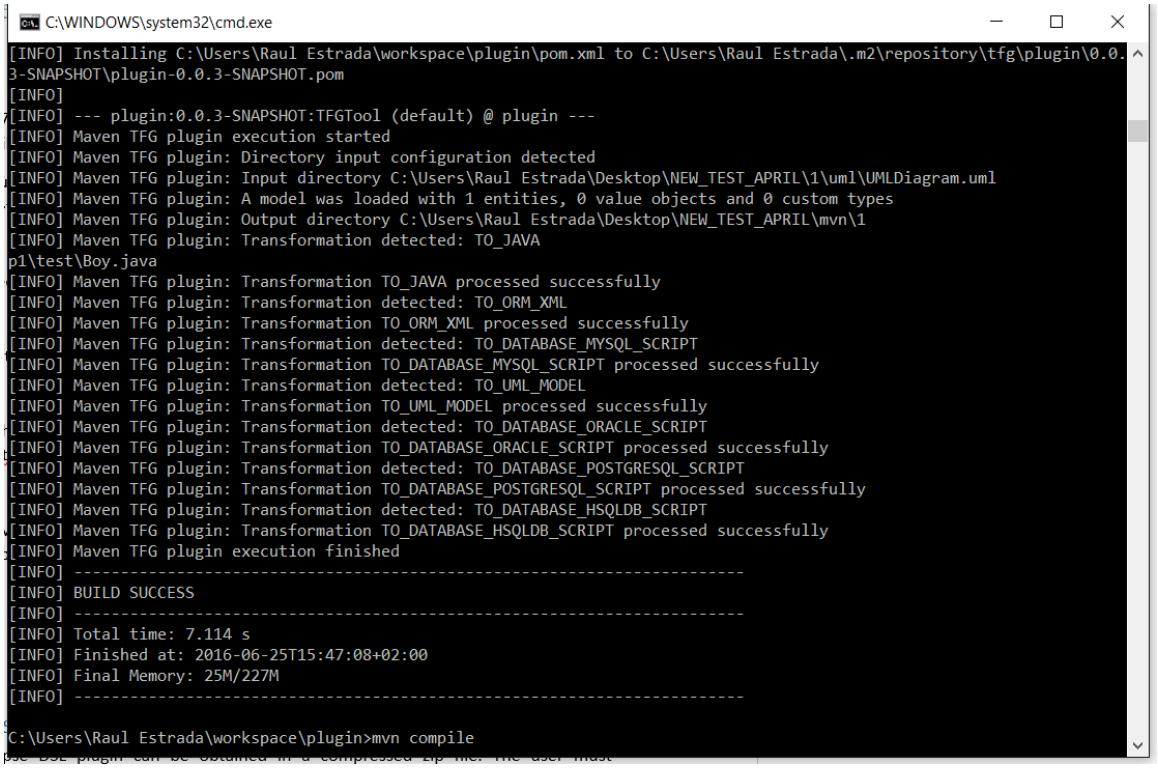

Figure 119. Maven plugin: Load from UML and generations

Figure 120 shows an example of a configuration to load the domain model from a database.

```
<execution>
    <phase>install</phase>
    <goals>
        <goal>TFGTool</goal>
    </goals>
    <configuration>
        <toolConfig>
            <databaseInput>
                <host>localhost</host>
                <port>5432</port>
                <name>RaulTFG</name>
                <username>raul</username>
                <password>tfg</password>
                <type>POSTGRESQL</type>
            </databaseInput>
            <toDirectory>C:\Users\Raul Estrada\Desktop\NEW_TEST_APRIL\mvn\1</toDirectory>
            <conversions>
                <param>TO_ORM_XML</param>
                <param>TO_UML_MODEL</param>
                <param>TO_DATABASE_MYSQL_SCRIPT</param>
                <param>TO_DATABASE_ORACLE_SCRIPT</param>
                <param>TO_DATABASE_POSTGRESQL_SCRIPT</param>
                <param>TO_DATABASE_HSQLDB_SCRIPT</param>
                <param>TO_JAVA</param>
            </conversions>
        </toolConfig>
```

Figure 120. Maven plugin: configuration to load model from database

4.4. Eclipse DSL plugin manual

The Eclipse DSL plugin can be obtained in a compressed zip file. The user must uncompress it and then go to his/her Eclipse windows. In Eclipse, go to the *Help* menu in the menu bar. Then, access the *Install New Software…* menu item. A new window will be displayed. Click the *Add…* button as shown in Figure 121.

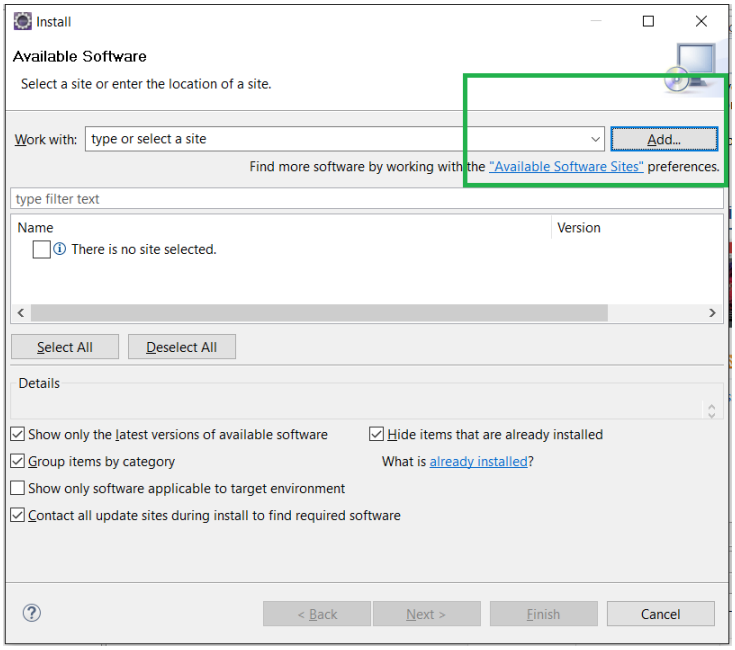

Figure 121. Eclipse DSL plugin: install software

A new window will open, the user must click the *Local…* button, select the new uncompressed file with the Eclipse DSL plugin and accept. Back in the *Install* window, the new available software will be displayed and ready to be installed.

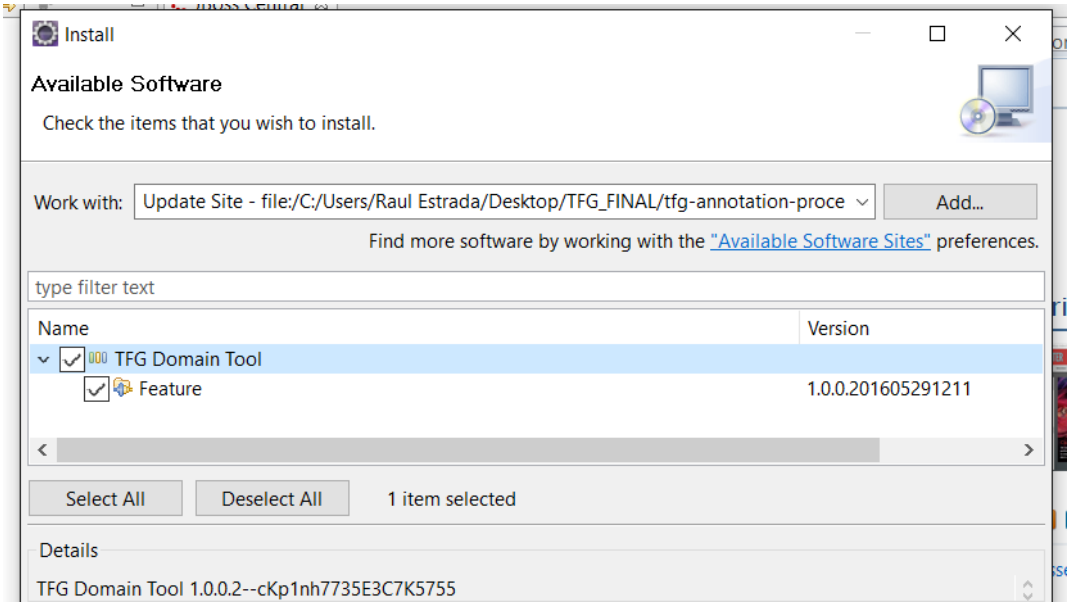

Figure 122. Eclipse DSL Plugin installation

Once installed, whenever a new file with the .tfg extension is created, the Eclipse environment will ask the user if he/she wants to apply the DSL editor. If the user accepts, the DSL editor is launched and developers and domain experts can benefit from its features, such as autocomplete and early compilation error detection, as shown in Figure 124 and Figure 125.

4.5. Eclipse product manual

The Eclipse product will launch a new Eclipse workbench instance, with the basic features and the custom TFG DSL editor already installed. The product is downloaded as a zip file, and the user will simply uncompress such file. The uncompressed directory will contain two folders: "repository" and "eclipse". In the latter the user can find an application executable file named "*eclipse*".

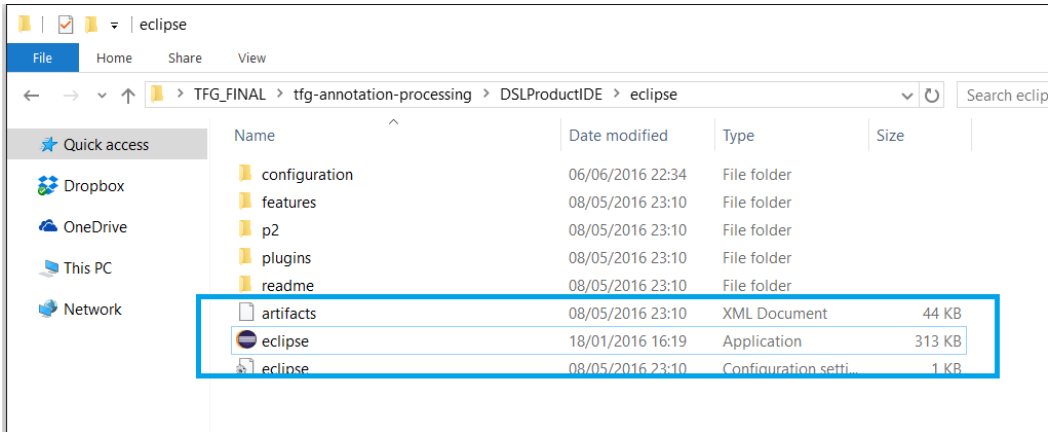

Figure 123. Eclipse product: eclipse executable file

Executing the previous file will open a new Eclipse window. Then, once a file with the .tfg extension is created, the Eclipse environment will ask the user if he/she wants to apply the DSL nature. The user should accept and say "yes" so that the DSL editor will be applied. From that moment forward, the editor is applied and the user can express the essence of a domain model using this editor and custom features such as autocomplete and compilation error detection (which allows to detect invalid syntax or structure, for instance).

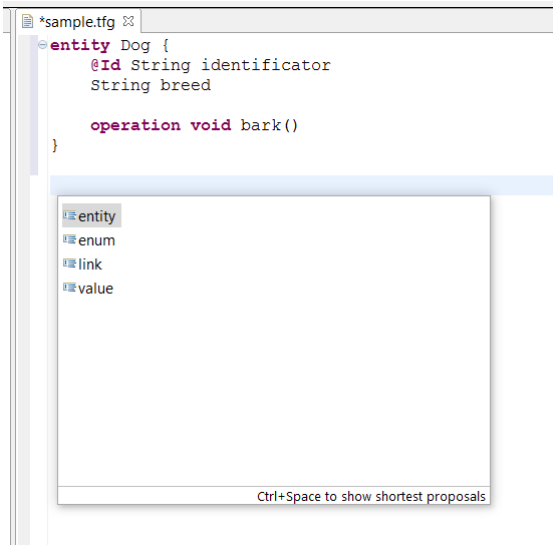

Figure 124. Eclipse product: autocomplete feature

Figure 124 shows the autocomplete feature of the DSL editor in action. Figure 125 shows a compilation error detected and displayed in the editor, since "*entities*" is not a valid keyword.

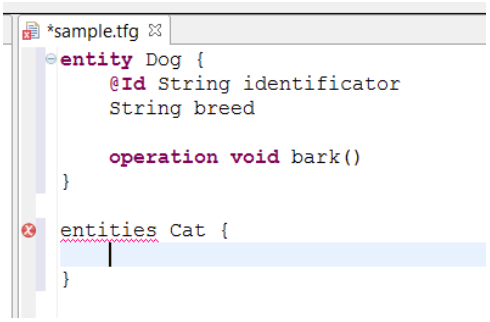

Figure 125. Eclipse product: compilation error detection

4.6. Developer manual

In order to recreate the development environment so that a developer can work, perform modifications or extend the system, the steps presented in this manual should be followed.

4.6.1. Java Development Kit

First, the Java Development Kit needs to be installed. The user can access the Oracle webpage[10] to download it for free. Once installed, the user needs to create a new system environment variable called *JAVA_HOME* whose value points to the directory where the JDK has been installed. Something like the following:

[10] http://www.oracle.com/technetwork/java/javase/downloads/index.html

```
JAVA_HOME          C:\Program Files\Java\jdk1.8.0_60
```

Then, to check if the installation and environment value have been correctly installed and configured, open a command-line or terminal and write:

```
javac -version
```

The previous command should print information about the version of the JDK installed.

Figure 126. Developer manual: install JDK

4.6.2. Eclipse

Eclipse was the IDE (*Integrated Development Environment*) used to program and develop this project, so to recreate the development environment, this program needs to be installed as well. The user can download it for free at the Eclipse official webpage[11]. Many tools are offered, but the user needs to download the Eclipse IDE (Eclipse Mars was used for this project).

[11] https://www.eclipse.org/downloads/

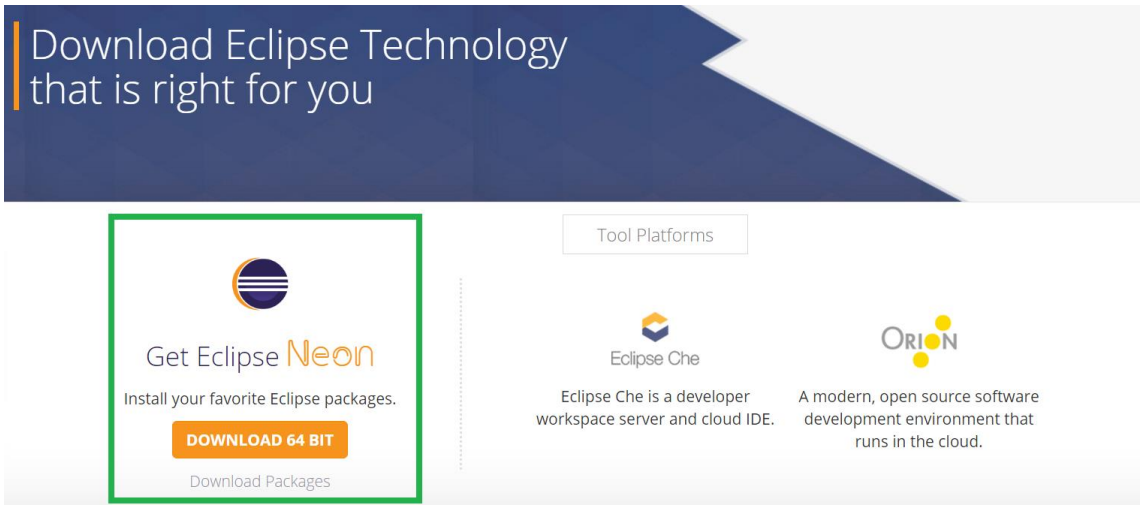

Figure 127. Developer manual: Install Eclipse

4.6.3. Maven

Maven is a build automation tool that handles the different dependencies a project has, and performs different tasks when a project is built. Maven needs to be installed to work with the Maven plugin project.

The user can download Maven for free from the Apache Maven Project website[12]. One of the Maven requirements is to have a JDK installed, so this software program should be installed after the JDK.

In the installation page, the user can choose to download a binary zip archive. Then, uncompress the folder and move it to a location the user considers appropriate for a program directory. Then, the user needs to go to the Environment Variables configuration and edit the *PATH* variable to add the directory path of the bin folder in the Maven root directory. Something like the following like `D:\Programs\apache-maven-3.3.9\bin`.

Figure 128. Developer manual: Download Maven

[12] https://maven.apache.org/

Finally, to check Maven has been installed and configured successfully, the user can open a command-line or terminal and execute: `mvn -version`. This should print information about the Maven program installed.

4.6.4. XText

XText is an Eclipse project and the framework used to design and implement the custom DSL for the project. Installation instructions are provided in the XText official webpage[13].

The user must open Eclipse. Then, in the menu bar go to *Help* and select the *Install New Software…* option. Go to the XText official webpage link provided above and copy one of the URLs for the Eclipse installation (*Releases, Milestones* or *Nightly Builds*). In Eclipse, in the new dialog window paste the URL in the *Work with* field. Then, select XText, click Next and finish the installation of the new software in your Eclipse environment.

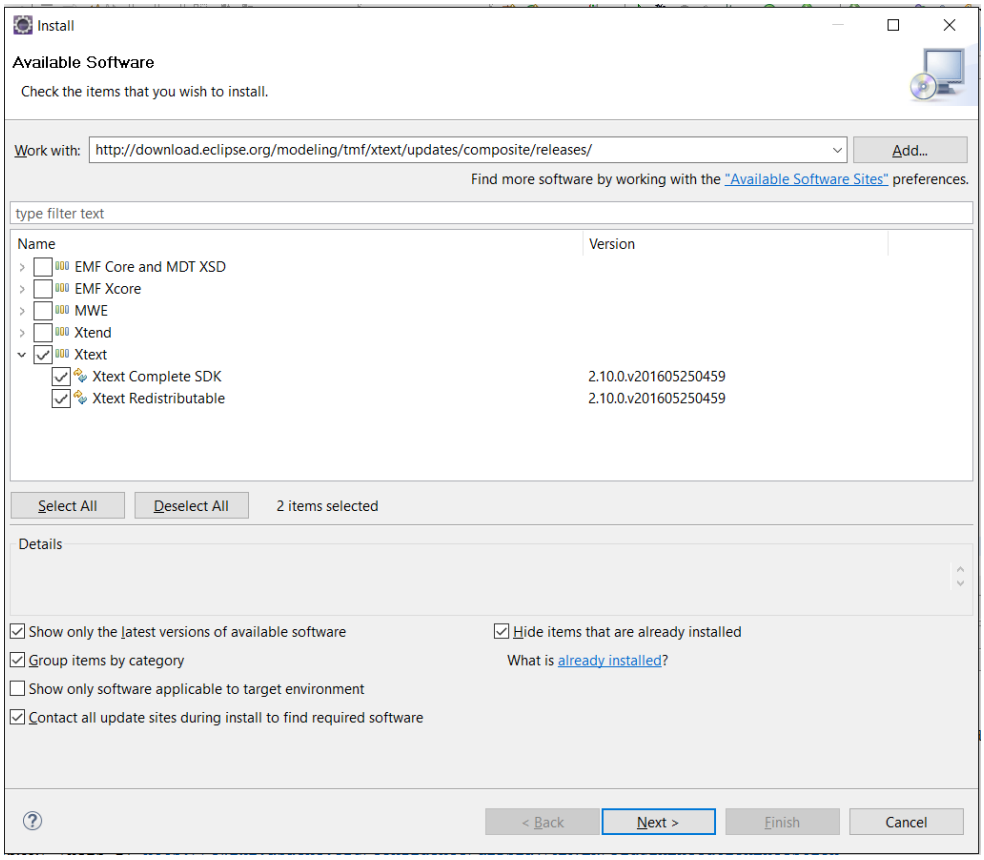

Figure 129. Developer manual: Install XText

4.6.5. Databases

Since the domain tool provides the option to load a model from databases and generate database scripts, the different databases supported (MySQL, PostgreSQL, HSQLDB and Oracle) were installed.

[13] http://www.eclipse.org/Xtext/download.html

PostgreSQL database management system can be downloaded at its official webpage[14].

MySQL Enterprise Edition can be downloaded at the MySQL official webpage[15].

HyperSQL database management system can be downloaded from its official webpage, and a compressed zip file can be downloaded here[16].

Oracle Database Express Edition can be downloaded from its Oracle official webpage[17].

4.6.6. Work with the Project

The different projects developed for the different products can be downloaded in a compressed zip file. When uncompressed, these projects can simply be imported in the Eclipse environment. In the menu bar, go to the *File* menu and select the *Import…* option. In the new dialog window, select the *Existing Projects into Workspace* in the *General* category, and select the directory containing the project to be imported.

[14] https://www.postgresql.org/download/
[15] https://www.mysql.com/downloads/
[16] https://sourceforge.net/projects/hsqldb/files/
[17] http://www.oracle.com/technetwork/database/database-technologies/express-edition/downloads/index.html