electronics

MDPI

Article

Design and Implementation of a Metadata Repository about UML Class Diagrams. A Software Tool Supporting the Automatic Feeding of the Repository

Paolino Di Felice [1,*], Gaetanino Paolone [2], Romolo Paesani [2] and Martina Marinelli [2]

[1] Department of Industrial and Information Engineering and Economics, University of L'Aquila, 67100 L'Aquila, Italy
[2] Gruppo SI S.c.a.r.l., 64100 Teramo, Italy; g.paolone@softwareindustriale.it (G.P.); r.paesani@softwareindustriale.it (R.P.); m.marinelli@softwareindustriale.it (M.M.)
* Correspondence: paolino.difelice@univaq.it; Tel.:+39-320-423-2540

Abstract: Model-Driven Engineering is largely recognized as the most powerful method for the design of complex software. This study deals with the automated archival of metadata about the content of UML class diagrams (a particularly relevant category of models) into a pre-existing repository. To define the structure of the repository, we started from the definition of a UML metamodel. From the latter, we derived the schema of the metadata repository. Then, a parser was developed that is responsible for extracting the useful information from the XMI file about class diagrams and enters it as metadata into the repository. The parser has been implemented as a Java web interface, while the metadata repository has been implemented as a PostgreSQL database based on the JSONB data type. The metadata repository is thought to support modelers in the initial phase of the process of the development of new models when looking for artifacts to start from. The schema of the metadata repository and the Java code of the parser are available from the authors.

Keywords: MDE; UML; class diagram; metadata; repository; NoSQL database

Citation: Di Felice, P.; Paolone, G.; Paesani, R.; Marinelli, M. Design and Implementation of a Metadata Repository about UML Class Diagrams. A Software Tool Supporting the Automatic Feeding of the Repository. *Electronics* **2022**, *11*, 201. https://doi.org/10.3390/electronics11020201

Academic Editor: Juan M. Corchado

Received: 24 October 2021
Accepted: 29 November 2021
Published: 10 January 2022

1. Introduction

Time and quality are critical factors in the development of complex software projects, so it is necessary to enforce consistent reuse to increase the quality while reducing the development time. Regarding programming, reuse has been successful for decades. Nowadays, however, software development is moving in the direction of modeling while the code is largely generated. The more the development of complex software is based on modeling, the more models become of paramount importance.

In the context of software application development based on Model-Driven (System) Engineering (MD(S)E) [1], UML models (in particular, class models and use case models [2,3]) are the artifacts to be reused. The models stored inside a company's folders represent valuable information since they capture domain knowledge. This is the reason that locating such artifacts can help new modelers to become familiar with recurrent modeling patterns and best practices within their business context. In [4], it is claimed that the support for finding and reusing modeling artifacts is still limited. Consequently, developers too often have to develop artifacts from scratch.

Repositories are the precondition for reuse, as highlighted in many studies (e.g., [4–9]). Ref. [4] provides a list of the most representative repositories. In this study, we assume that the corporate repository is structured as three independent but interrelated components:

- a repository about the modeling artifacts, briefly the Model Repository [10], developed in previous projects;
- a repository about the generated code;
- a repository containing metadata about modeling artifacts, briefly the Metadata Repository [10].

The design of the metadata repository about UML class diagrams (which are a relevant specification technique to describe the structure of the software system to be developed) is the focus of the first part of the present paper. A metadata repository can be defined as a shared database of information regarding engineered artifacts [9]. The metadata in the database describe the class diagrams, besides providing the link to those artifacts inside the model repository. Our metadata repository is structured as a PostgreSQL database.

Once a metadata repository about UML artifacts is made available, a relevant, time-consuming, and monotonous task is to feed it. The second part of this paper describes a parser that makes such a step automatic. This accomplishment was possible because UML models can be saved into a CASE tool format (e.g., StarUML) and then into the XMI format. These files contain *all* model information (e.g., class names, attributes, operations, association ends, multiplicities, association names, etc.). By processing them, it is possible to find the information to be stored in the metadata repository. In the present version of the parser, the extraction of the content of an XMI file and its formatting as NoSQL records works for XMI files coming from StarUML (https://staruml.io/download, accessed on 6 September 2021). The extension of this software component to other UML tools is planned as future work.

The paper is structured as follows. Section 2 describes our approach. Section 3 presents the UML metamodel of the metadata repository about class diagrams, while Section 4 details the storage technology for implementing it. Section 5 depicts the paths to feed the metadata repository. Then, it focuses on the software module, which takes as input the XMI file about a single class diagram present in the model repository and finds the data to be stored in the metadata repository. These metadata provide a detailed description of the class diagram. Section 6 proposes a simple case study; Section 7 recalls previous studies similar to ours; Section 8 concludes the paper and outlines the future work.

Three appendices integrate the information given in Sections 4–6. In detail, Appendix A contains the SQL scripts for the creation of the database tables and shows the actual implementation of the OCL constraints that complete the description of the metamodel of Section 3. Appendix B shows an excerpt of the content of the database concerning the information coming from the case study. Appendix C simulates a session of interaction with the metadata repository by a software engineer looking for class diagrams to be reused. The queries are general, so they can be reused by the modelers.

2. Our Approach

In [11], the authors provide an overview of the objectives, beneficiaries, architecture and technologies of an ongoing industrial project whose goal is to release an open-source software tool (called `xMetaRep`) devoted to the creation, feeding and querying of a metadata repository about UML class diagrams. The project comprises two distinct actions: the first belongs to the conceptual level, while the second one belongs to the technological level (Figure 1). Action 1 involves the design of the metadata repository about class diagrams, while Action 2 concerns the development of the user interface (i.e., `xMetaRep`) on top of it.

2.1. Action 1

The design of the metadata repository starts from a general UML metamodel of the repository. The latter is then mapped into the corresponding Entity–Relationship (E–R) conceptual metaschema (Figure 2). The translation of such a schema produces the logical schema of a relational database.

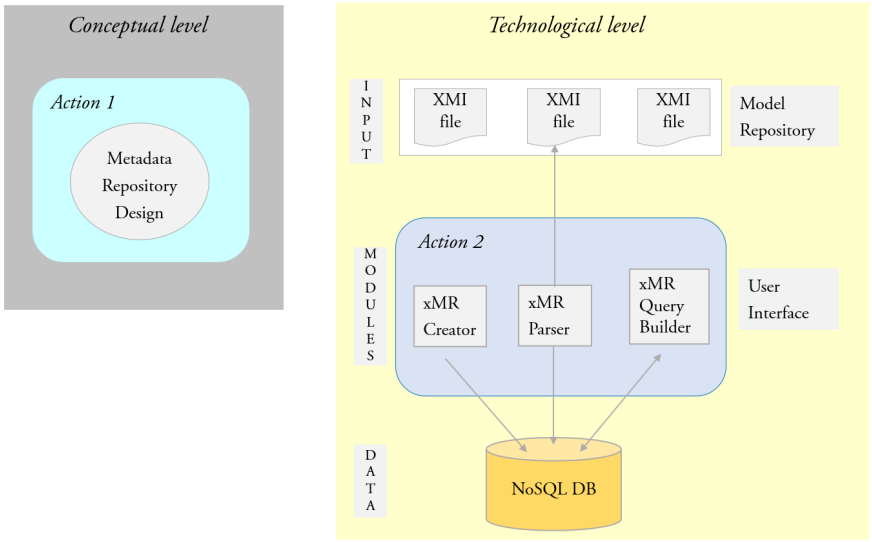


Figure 1. Overview of our approach [11].

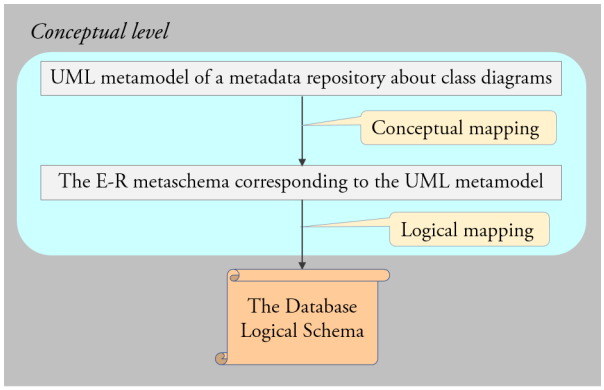


Figure 2. Design activities at the conceptual level.

As pointed out in [10], the preliminary task of a repository architect is to choose the storage technology for the metadata repository. In this paper, we structured it as a PostgreSQL database. The PostgreSQL Object Relational Database System is an open-source mix of relational and NoSQL databases; in fact, it has supported, for many years, document databases and key–value databases—two of the most common NoSQL database types. This is the reason that, in the EnterpriseDB white paper [12], the expression "Postgres NoSQL" is used. In line with [12], in our paper, we state that our metadata repository is a NoSQL database because it is composed of tables comprising attributes of the JSONB data type. In detail, the work identifies the number and the schema of the tables composing the metadata repository. The schema of the tables is independent of the internal organization of the classes inside the class diagram. This result is brought about by the flexibility of the JSONB data type. Several advantages come from storing metadata about UML models within a company NoSQL repository. First, NoSQL databases overcome "pure" relational ones in terms of flexibility. Second, NoSQL databases guarantee a high level of interoperability. Third, a company database ensures the cooperation of modelers and developers in the development of a system. Fourth, querying the metadata inside the company's repository helps in the reuse of the artifacts that best fit the requirements of new projects. Fifth, studying UML diagrams from previous high-quality projects can help novice modelers to learn from the experience of senior ones. This latter motivation has been pointed out by Gosala et al. in [8]. The availability of a centralized company repository containing metadata about UML diagrams implemented as a NoSQL database represents the ultimate

alternative to the current scenario, where software engineers have to manually search these artifacts inside a huge number of files stored in different folders (i.e., the model repository).

2.2. Action 2

The field of modeling repositories addresses mostly collaborative work (e.g., [13–15]). The Repository for Model-Driven Development (ReMoDD) [6,7] is a project from industry and academia aiming at developing a public resource collecting artifacts coming from high-quality MDD experiences (ReMoDD is located at the following URL: http://www.cs.colostate.edu/remodd/, accessed on 11 September 2021). The objective of the project is to facilitate the sharing of relevant knowledge and experience for improving MDD research productivity and education. In the collaborative domain, a relevant issue concerns the management of the versioning of artifacts during their development. This aspect is marginal in our reference scenario delimited by the firm's boundaries. In this context, we assume that single modelers undertake the development of UML diagrams (e.g., use case diagrams, class diagrams, sequence diagrams, etc.) for specific software projects. At the end of the development process, it is a modeler's responsibility to invoke the archiving, in the company repository, of the metadata about the developed diagrams.

Figure 1 (right side) shows the components of `xMetaRep`:

- The Model Repository of the company contains the XMI files about UML class diagrams. These files are the input data for the overall process for building and feeding the metadata repository about such a category of artifacts.
- The User Interface is composed of three software components: `xMR Creator`, `xMR Parser` (in [16], `xMR Parser` is called `XMI_to_Parser`) and `xMR Query Builder`. They support, in turn, (a) the creation of an empty instance of the NoSQL DB; (b) the extraction of metadata from the XMI files and the copying of them into the NoSQL DB; (c) the instantiation of a predefined set of flexible query templates against the NoSQL DB.
- The NoSQL DB layer denotes the Metadata Repository about UML class diagrams in the Model repository.

The parser has been implemented as a Java web application that adheres to the Model-View-Controller (MVC) pattern. The Spring framework was used for creating the application. Spring is the most adopted Java framework worldwide. The core of the web application is `xMR Parser`. It is the result of a long and engaging journey that started in 2008 [17], in which research, implementation and validation in the domain of enterprise web applications have been joined together. In 2020, Paolone et al. [16] introduced an automatic process to develop such a category of web applications. The frame of reference is MDA, and the pillars of the proposal are use cases, class and sequence diagrams. These diagrams cover, in order, the structure and the behavior of the system to be developed, as well as their interactions. In this way, all the system requirements that the OMG recommends are satisfied. The methodological process ensures continuity between business modeling, system modeling, design and implementation. This lays the foundation for the mapping of the behavioral business model into a consistent software that meets the requirements. A proprietary Java technology platform called `xGenerator` implements the Software Development Process described in [16]. At a high level of abstraction, such a tool acts as a black box that receives as input a UML model and returns the Java code of the web application. `xMR Parser` is a component of `xGenerator`.

3. UML Metamodel for a Metadata Repository about Class Diagrams

The basic elements that determine the structure of a metadata repository about class diagrams are collected in the UML metamodel of Figure 3. The metamodel shows that the class diagram, which models the *structural view* of a completed software project, is stored inside a company's package, possibly nested. Each diagram comprises a set of classes linked through several different kinds of relationships; moreover, classes may be linked through a generalization hierarchy, while association classes are a specialization of the

notion of class. Each association is described in terms of two or more association ends; an association end is bound to a class. The metamodel of Figure 3 depicts the meta-associations between the UML elements `class` and `generalization` as a two-way relationship, for the reasons explained in [18].

The metamodel of Figure 3 introduces two variants to standard UML class diagrams: operations constitute part of the class description with their signature; moreover, the data types of the attributes are made explicit. Unlike previous studies, whose purpose was to build a repository that offers support for checking the structural consistency of the class diagram during the process of modeling of a software product (e.g., [19,20]), in our work, the emphasis is on providing support to the modelers before they start the modeling stage of the new software product. In this phase, it makes sense to investigate whether there are artifacts (in the model repository of the company) from which to start. In this working scenario, the more metadata about classes in the class diagram, the more effective and aware the decision regarding what to start from will be. Moreover, in the Lindholmen database schema (http://models-db.com/oss/default.aspx, accessed on 18 September 2021), the metadata about UML classes include attributes' data type, methods and their signature. Such a dataset includes links to more than 93,000 UML files spread across more than 24,000 GitHub repositories [21].

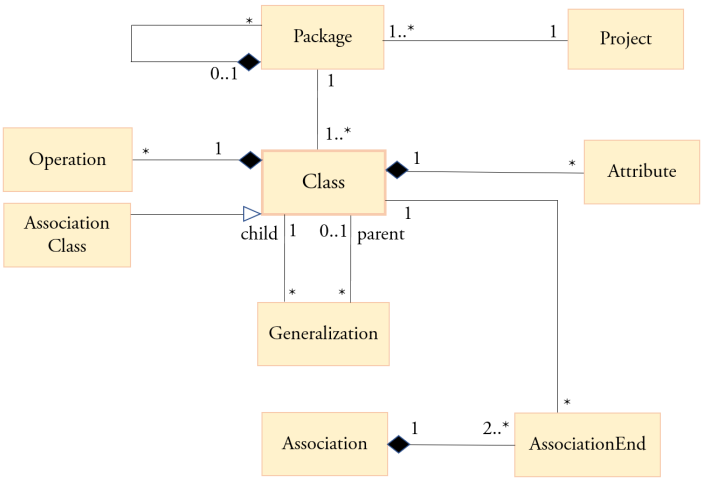


Figure 3. UML metamodel of a metadata repository about class diagrams.

According to the OMG conventions, if the association ends of a class are not explicitly named in the class diagram, the implicit rule is that their names are given as follows: ([2], p. 202): `"end"<class-name>` (Figure 4 proposes an example). Moreover, if the associations are not explicitly named, then their given names come from the following rule ([2], p.19):

`"A_"<association-end-name1>"_"<association-end-name2>`,

where `<association-end-name1>` is the name of the first association end and `<association-end-name2>` is the name of the second association end (see Figure 4).

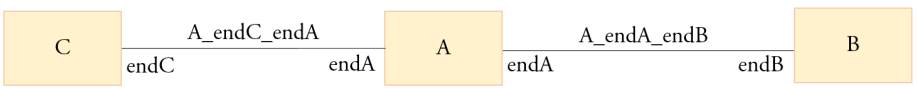


Figure 4. Examples of OMG naming conventions.

UML diagrams cannot by themselves provide all relevant aspects of a specification. To add more semantics to the metamodel of Figure 3, a set of integrity constraints, which each state of the information base must satisfy, is to be used. The OMG Object Constraint Language (OCL) is used to describe expressions on UML models [22]. Many studies have pointed out the relevance of adding OCL constraints to UML models for controlling the

correctness of their structure (e.g., [19]). In the present study, we focus on two categories of constraints. The first category must be satisfied by each class instance (briefly, the intra-UML-element constraint), while the second one must be satisfied by pairs of class instances (briefly, the inter-UML-element constraint). Of course, many more constraints need to be taken into account. For example, the OCL expression of Equation (1) formalizes the following invariant constraint: "a root class has no parent" (an invariant OCL expression must be true for all instances that it refers to at any time).

```
context class inv:self.isRoot implies self.generalization->isEmpty;    (1)
```

4. The Schema of the Metadata Repository

As pointed out in [10], the preliminary task of a repository architect is to choose the storage technology for the metadata repository. In this paper, we structured it as a NoSQL database. NoSQL is an umbrella term for different technologies. The most mature are known as: key–value databases, document databases, column databases and graph databases. Key–value databases are the simplest NoSQL databases. They store a set of key–value pairs. Redis implements this model. Document databases contain key–value pairs, which can be any sort of value, array or even another document. MongoDB implements this model. Column databases organize data into columns rather than rows; however, they operate in the same manner as tables do in relational databases. Apache Cassandra implements this model. Graph databases are for general-purpose use, particularly with unstructured data and social networks. Neo4j is a popular system example.

To build the metadata repository about class diagrams, the flexibility demonstrated by document databases is fully satisfactory; in fact, it makes the database schema independent of the internal organization of the classes in the class diagram. Specifically, flexibility is needed to model the names and data types of attributes and the names and I/O parameters of the operations. Both these features are highly variable moving from one class to another. We took into account two alternative open-source technologies: MongoDB and PostgreSQL (from version 11.0, the latter system fully supports the storage of documents in the JSONB format, as MongoDB does). The final choice was to adopt PostgreSQL because it closed the gap that motivated the rise and development of NoSQL technologies; moreover, PostgreSQL provides capabilities that NoSQL technologies simply cannot, namely a powerful query language, a sophisticated query optimizer, data normalization, joins and referential integrity. A positive consequence of the availability of a powerful query optimizer is that PostgreSQL outperforms MongoDB in almost all cases, as has been pointed out in recent studies (e.g., [23,24]). For example, in [24], the authors loaded a dataset of 200 million records of JSON documents in MongoDB (Community Server1, ver. 4) and in PostgreSQL (ver. 11) using the JSONB data type. The aim of the experiment was to compare the performance of the two systems on four custom-written queries over one year of GitHub archive data. PostgreSQL was found to be between 35 and 53% faster on three of the four queries, and 22% slower on the other one.

As the next step, repository architects have to decide whether to add version metadata or not [10]. As explained in Section 1, our solution does not include metadata about versions of class diagrams simply because only their final version is stored in the model repository. Moreover, affiliation information about the artifact owners, which might be relevant in the case of collaborative repositories [10], is not necessary for corporate metadata repositories.

In order to determine the schema of the metadata repository, the UML metamodel of Figure 3 was mapped to the corresponding Entity–Relationship (E–R) schema (Figure 5). The translation of such a schema gives rise to the seven tables listed below (underlined attributes denote the primary key). Appendix A contains the SQL scripts for the creation of these tables.

- project(projectID, name);
- package(packageID, name, URI, projectID);
- class(classID, name, packageID, associationClass, parent);

- `operation(classID, list);`
- `attribute(classID, list);`
- `association(associationID, name);`
- `associationEnd(associationEndID, name, classID, associationID).`

The `Primary Key` constraint is the mechanism offered by the database technology for implementing OCL intra-UML-element constraints. Seven `Primary Key` constraints have to be defined, one for each class.

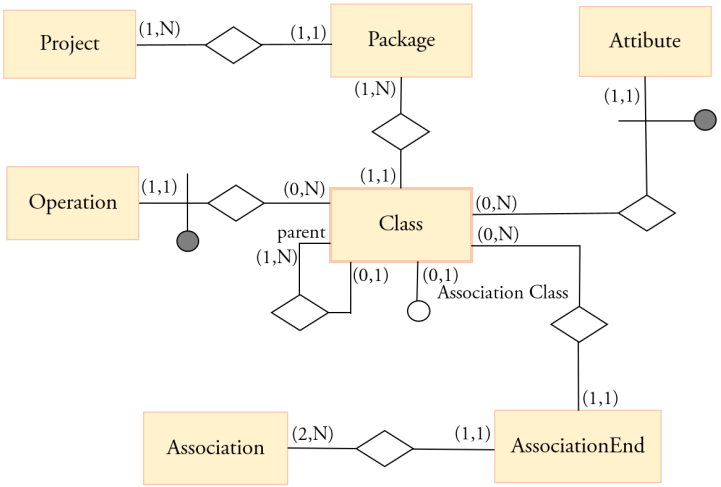


Figure 5. The E–R metaschema corresponding to the metamodel of Figure 3.

For example, the OCL expression of Equation (2) states that *all* instances of the `class` classifier must have a distinct value for the property `classID`.

```
context class inv UniqueClassID:
                    self.allInstances() implies isUnique(ClassID);   (2)
```

As is well-known, the relationship between classes is implemented as a referential integrity constraint between the foreign key of the *slave* table and the primary key of the *master* table. This mechanism of the relational database technology is the simplest means of implementing OCL inter-UML-element constraints. For example, the OCL expression of Equation (3) states that *each* instance of `class` must be linked to a unique value of `packageID`, i.e., to a unique physical package.

```
context class inv UniquePackage:
                    self.allInstances() implies isUnique(packageID);   (3)
```

Appendix A shows the actual implementation of the OCL constraints mentioned above.

5. Architecture and Implementation of the XMI Parser

Figure 6 depicts the paths to feed the metadata repository about UML artifacts. The starting point for collecting metadata about UML class diagrams is twofold. One entry point is represented by an image of the diagram, while the alternative is represented by the file generated by one of the available UML tools (e.g., StarUML).

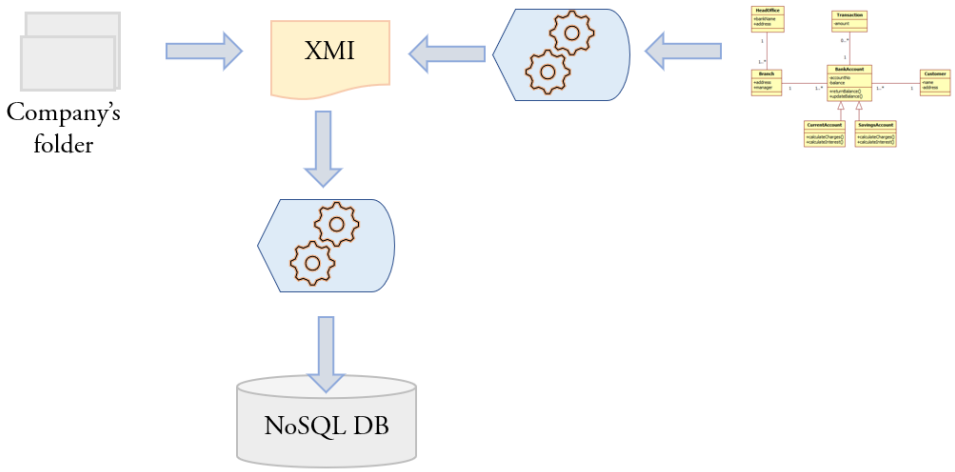


Figure 6. Paths to feed a metadata repository about UML class diagrams.

A few years ago, Robles et al. [25] mined the content of 24,717 different GitHub repositories, looking for UML models. They found 93,596 UML models, of which approximately 62% were images, the rest being either XML or UML files. This finding tells us that UML models are mostly stored as images in public repositories. To the best of our knowledge, the situation inside firms has not been investigated yet. If the starting point is an image of the class diagram, then a tool that automatically extracts the UML diagram from the image is needed. `Img2UML` is one such tool [26–28]. It is able to recognize shapes, symbols, lines and text; moreover, it identifies the role of diagram elements in the model. `Img2UML` returns as output a file in the XML Metadata Interchange (XMI) format. The authors claim that the class detection accuracy is 100%, relationship accuracy is 97% and symbol accuracy is 85%. To correct recognition mistakes in the returned UML class diagrams, the authors suggest importing the XMI file generated by `Img2UML` into StarUML and manually correcting the recognition mistakes.

Alternatively, metadata about UML class diagrams may come from XMI files about previous company projects. As explained in Section 1, we assume that the XMI files are in the model repository of the company ("Company's folder" in Figure 6). In both paths, the final step consists of migrating the information in the XMI file into the metadata repository. We have implemented this final step.

The idea of parsing the XMI file of the class diagram for extracting metadata describing its content is not new. For instance, in [29], Girgis et al. describe an XMI file parser that extracts metadata from this category of files to be used to calculate common metrics about the class diagram.

The parser has been implemented as a Java web application that adheres to the Model-View-Controller (MVC) pattern. The Spring framework (https://docs.spring.io/spring-framework/docs/4.3.x/spring-framework-reference/html/overview.html, accessed on 14 September 2021) was used for creating the application. Spring is the most adopted Java framework worldwide. Figure 7 shows the full stack of the Spring framework, while Figure 8 shows only the modules that were used in the development of the parser.

It has been remarked (see, for instance, Ref. [30]) that carrying out a project based on Spring is not a trivial task and, in fact, it takes a lot of time, even for small applications. This is a direct consequence of the large number of XML configuration files that have to be properly set up, so that the individual components and application modules might work properly. In order to simplify the process, besides Spring, we used also Spring Boot. Spring Boot is an addition to the Spring platform that makes it very easy to get started with the tool and create stand-alone, production-grade applications. In other words, Boot is not intended to replace Spring, but to make its operation faster and easier. According to a 2021 public survey carried out by JRebel [31] (from August through November 2020 among

876 members of the Java development community), 62% of the respondents were working with Spring Boot.

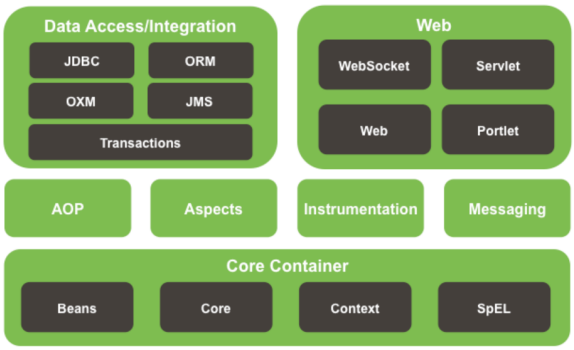


Figure 7. The stack of the Spring framework.

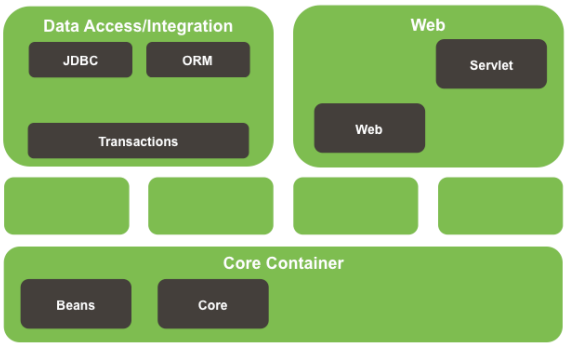


Figure 8. The modules of the Spring framework used to develop the parser.

The code of the parser inside the `main` folder is structured as shown in Figure 9. The class `ParsingApplication.java` (annotated as `@SpringBootApplication`) is the entry point of the Spring Boot application. Overall, the web application consists of 20 classes, 15 interfaces and 2 views, as described below:

- The MVC controller layer corresponds to the `controller` package. It contains 1 class that is responsible for loading the Java Server Page (JSP) views, handling events, user actions and the navigation logic.
- The Model layer corresponds to the `model` package, which contains 7 classes in the `entity` subpackage and 1 class in the `dto` subpackage. The `entity` package contains Plain Old Java Objects, which correspond in number and structure to the database tables (Section 4). For instance, `AssociationEntity` corresponds to the `association` table.
- The Repository layer corresponds to the `repository` package. It contains 7 interfaces, one for each entity class of the `model` package, which extend the `CRUDRepository` interface of Spring.
- The Service layer corresponds to the `service` package. It is structured as 8 interfaces and 8 classes; the latter implement the former. The classes use the relative interfaces of the repository to query the database. The `ParsingServiceImplementation` class uses the services of the other classes to insert the metadata extracted from the XMI file, as a single database transaction.
- The 2 utility classes contain constants and variables used to carry out the needed checks.
- The `webapp` package collects 2 views. They are 2 JSPs that implement the user interface of the web application. The first JSP is responsible for the loading of the XMI file from the user; then, such a file is passed to the controller for the parsing (Figure 10); meanwhile, the second one shows the result (either "Success" or "Error").

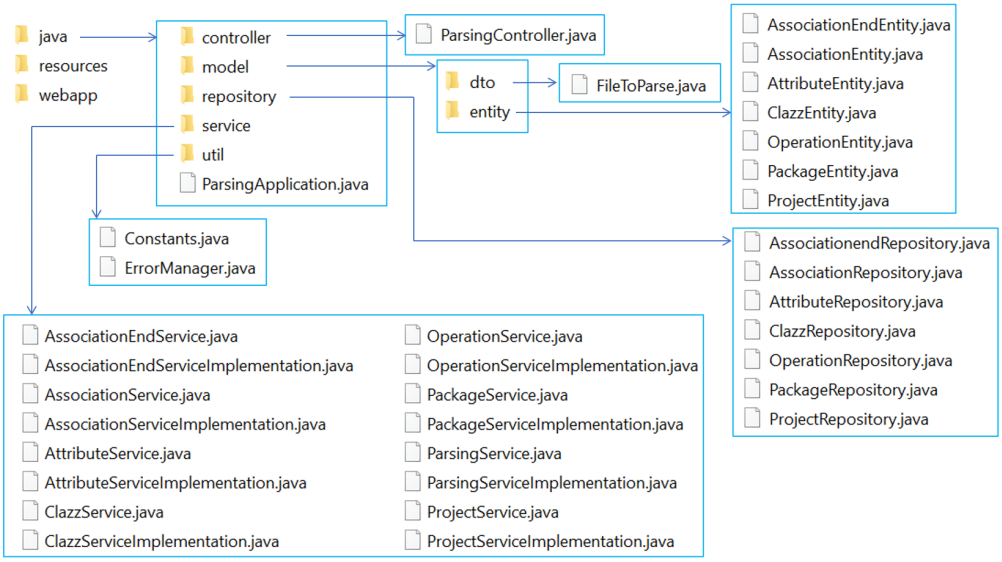


Figure 9. The organization of the source code of the parser.

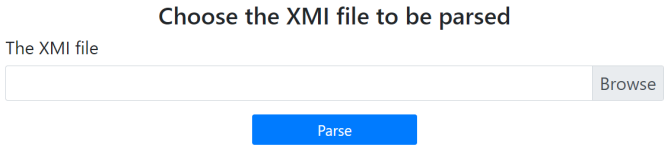
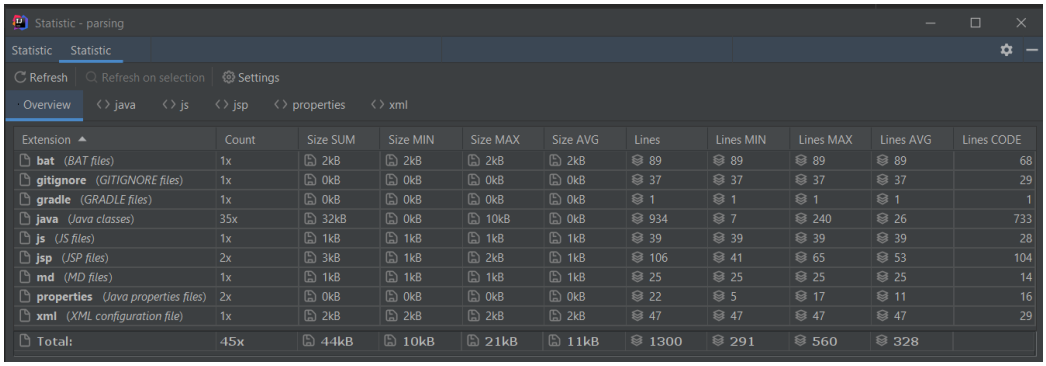


Figure 10. The input screen of the parser.

The screenshot in Figure 11 collects the statistics of the parser web application. Files `bat`, `gitignore`, `gradle`, `md` and `properties` are auto-generated by the `Intellij Idea` IDE when the software project is created; the remaining files (`java`, `js`, `jsp` and `xml`) were written by us. The second column of the figure counts the number of files of a given category in the web application. For instance, we can see that there are 35 Java classes (Figure 9 confirms such a number). The total number of lines of CODE is 894 (blank lines excluded), while 128 is the total number of autogenerated LOC.



Extension	Count	Size SUM	Size MIN	Size MAX	Size AVG	Lines	Lines MIN	Lines MAX	Lines AVG	Lines CODE
bat *(BAT files)*	1x	2kB	2kB	2kB	2kB	89	89	89	89	68
gitignore *(GITIGNORE files)*	1x	0kB	0kB	0kB	0kB	37	37	37	37	29
gradle *(GRADLE files)*	1x	0kB	0kB	0kB	0kB	1	1	1	1	1
java *(Java classes)*	35x	32kB	0kB	10kB	0kB	934	7	240	26	733
js *(JS files)*	1x	1kB	1kB	1kB	1kB	39	39	39	39	28
jsp *(JSP files)*	2x	3kB	1kB	2kB	1kB	106	41	65	53	104
md *(MD files)*	1x	1kB	1kB	1kB	1kB	25	25	25	25	14
properties *(Java properties files)*	2x	0kB	0kB	0kB	0kB	22	5	17	11	16
xml *(XML configuration file)*	1x	2kB	2kB	2kB	2kB	47	47	47	47	29
Total:	45x	44kB	10kB	21kB	11kB	1300	291	560	328	

Figure 11. The statistics of the parser web application.

At a high level of abstraction, the processing flow of the developed Spring MVC Web application (from receiving the modeler request till the response is returned) is shown in Figure 12.

The extraction of the metadata from the XMI file (i.e., class names, attributes, operations, association ends, multiplicities, association names, etc.) is done by `xMR Parser`, while their persistence is the responsibility of the Repository layer (Data Access) (Figure 12). `xMR Parser` is a component of the `xGenerator` proprietary software [16]; it has been wrapped inside the Service layer (which implements the Business Logic of the Web application) (Figure 12). `xMR`

`Parser` is an alternative to the well-known `Acceleo` (https://www.eclipse.org/acceleo/, accessed on 14 September 2021), an open-source solution. In our case, the adoption of `Acceleo` would have required much more time than `xMR Parser`, which, on the other hand, we were familiar with.

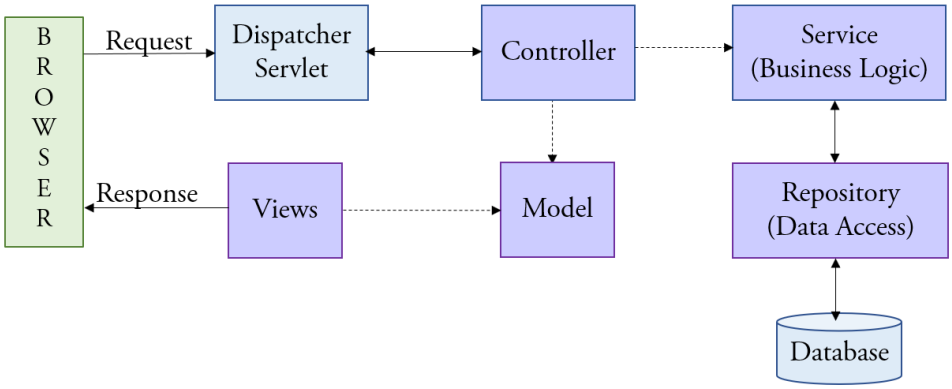


Figure 12. The flow of the parser web application. Violet rectangles denote classes implemented by us, while the blue ones denote components provided by the Spring framework.

The algorithm behind the parser works as follows. Preliminarily, checks are carried out on the existence of the file chosen by the modeler, in order to ensure that it has an XMI extension and that it is not empty. If the checks are successful, then the actual processing begins. The first step reads the input XML file and creates a tree-like structure, which resides in the main memory. This step is accomplished by the Java DOM parser. The visit of the tree data structure is organized as a cycle in which, at each iteration process, one of its "nodes" is searching for one of the six XMI types that we are interested in, namely (Figure 13): `uml:Package`, `uml:Class`, `uml:Association`, `uml:Attribute`, `uml:Operation`, and `uml:AssociationEnd`. At the end of the iteration, it is the responsibility of the `saveToDb()` method and the Hibernate ORM to perform the uploading of the metadata collected in the output list into the repository.

The used technologies to develop the web application are listed in Tables 1 and 2 and graphically summarized in Figure 14.

Table 1. List of the software technologies used.

Layer	Adopted Software Technologies
User Interface	`JSP, JavaScript, Bootstrap`
Business Logic	`Intellij Idea, Java, Spring, Spring Boot, Hibernate, JDOM Parser, xMR Parser`
Data	`PostgreSQL`

Table 2. Communication software between adjacent layers.

Adjacent Layers	Technology
User Interface–Business Logic	`JSP, Spring`
Business Logic–DBMS	`Spring, Spring Boot`
DBMS–DB	`Hibernate`

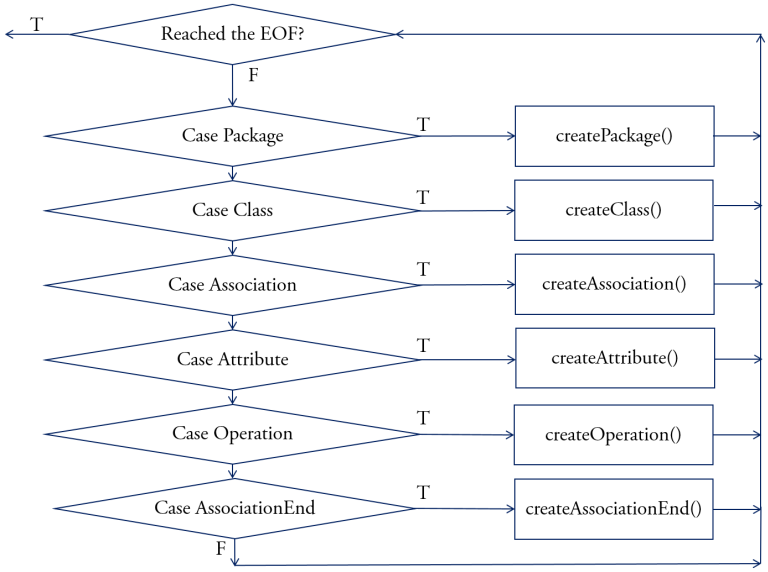


Figure 13. The processing of the tree-like structure. The flow chart reproduces the structure of the code (a "`switch`") and uses the same names of the methods in it. For example, `createClass()` creates an instance of a Java object, initializes its fields by means of getters and setters with the metadata fetched from the attributes of the current node and adds such an object to the output list.

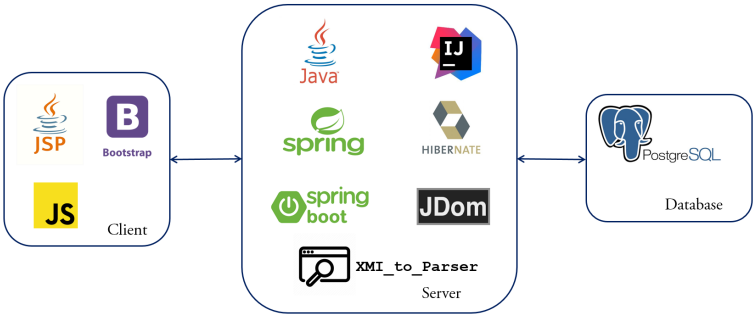


Figure 14. The mosaic of the used software technologies.

In summary, the automatic step that the parser is responsible for is fundamental for the success of the experiment of building and keeping updated a corporate metadata repository about UML class diagrams, since, as remarked, for example, in [32], one of the desirable properties of repositories is that their content does not depend on a single person or a small group of people.

6. Case Study

The example that we refer to migrates into the metadata repository information about the UML class diagram of Figure 15. The diagram comes from the `ATMProject` described in [33] (the full code of the Java classes of the project are available at https://www.softwareindustriale.it/atmproject.html). Table 3 provides a summary of the metadata that have to be stored into the database. In brief, only the names of the attributes and operations of the `customer` class are listed. Appendix B shows an excerpt of the content of the database as the result of the automatic parsing of the corresponding XMI file.

Appendix C simulates an interaction session with the metadata repository by a software engineer looking for potential class diagrams to be reused. Each interaction takes place as an SQL query against the sample database. Of course, the more metadata stored in the database, the more powerful queries can be written.

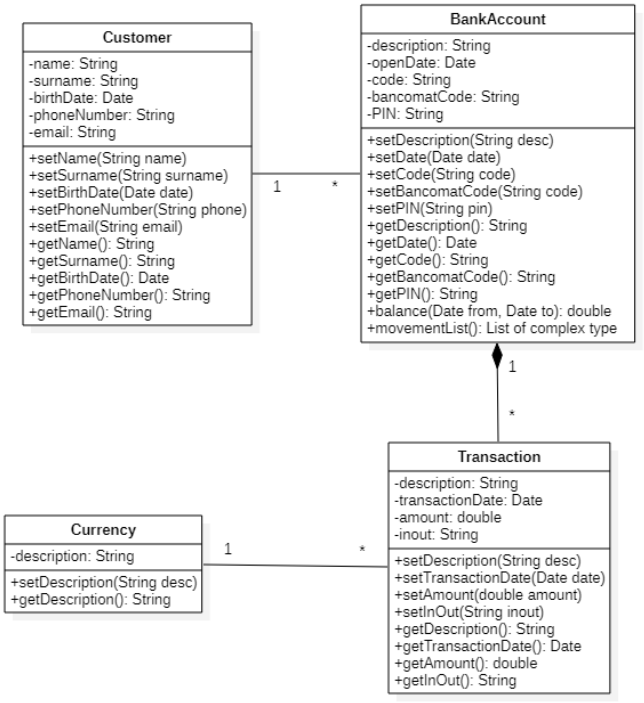


Figure 15. The UML class diagram of reference.

Table 3. An excerpt of the metadata to be inserted into the corporate repository.

Table	Number of Tuples	Metadata
project	1	`ATMProject`
package	1	`myUMLClasses`

association	3	`A_endCustomer_endBankAccount`
		`A_endBankAccount_endTransaction`
		`A_endTransaction_endCurrency`

class	4	`customer`
		`bankAccount`
		`transaction`
		`currency`

operation	32	`setName(String:name)`
		`setSurname(String: surname)`
		`setBirthDate(Date: date)`
		`setPhoneNumber(String: phone)`
		`setEmail(String: email)`
		`getName(): String`
		`getSurname(): String`
		`getBirthDate(): Date`
		`getPhoneNumber(): String`
		`getEmail(): String`

attribute	15	`name: String`
		`surname: String`
		`birthDate: Date`
		`phoneNumber: String`
		`email: String`

associationEnd	6	`endCustomer`
		`endBankAccount`
		`endBankAccount`
		`endTransaction`
		`endTransaction`
		`endCurrency`

7. Related Work

The idea of building a corporate repository about UML models is not new. For example, Belaunde describes a project started in 1996 at the France Telecom research center [34]. The aim of the project was the design and development of a repository about UML class diagrams. As in our case, the structure of the metadata repository comes from an UML metamodel, where the abstraction about "operations" is missed. The repository was implemented as a Java program. Therefore, while, in [34], each metaclass maps into a programming class, in our approach, each metaclass maps into a database table.

Ref. [5] discusses the architecture, the schema and the implementation of a repository at the design level, therefore called the "design repository". Such a repository is at the core of the SPOOL reverse software engineering environment. The design repository stores information about the source code of software systems, enabling users to conduct tasks of system analysis and reengineering. The SPOOL design repository was implemented as an object-oriented database.

In [13], Tran et al. adopt a NoSQL graph database to store UML artifacts. The graph represents each model (a UML class in the paper, but the authors claim that the solution is general) as a node, while edges between pairs of nodes express the kind of relationship between the classes. The final aim of their research was to build a model recommender system on top of the repository in order to support modelers during the modeling activities.

Ritter and Steiert [20] implemented a UML repository based on a metamodel and by exploiting an object-relational database management system. The mapping of the UML metamodel to the database schema is described in the paper. In order to enforce data integrity in the repository, the authors implemented OCL expressions as SQL constraints. This study was the first one exploiting the database technology for the purposes of building an UML repository.

In [27], the authors claim that each class model processed by the `Img2UML` tool is stored in a model repository implemented as a Microsoft Office Access 2010 database. Therefore, this approach does not adopt a metadata repository about class diagrams.

In [28] (see also http://models-db.com/oss/default.aspx, accessed on 5 September 2021), a relational database schema is described. It is devoted to collecting metadata about UML diagrams developed at the early stages of open-source projects. From the point of view of the metadata taken into account, the schema of our (metadata) repository overlaps with their schema significantly (Table 4), while the organization of the two databases is totally distinct, which is a consequence of the adoption, in our case, of the JSONB data type. This choice makes our solution more compact (seven tables vs. ten (http://models-db.com/oss/default.aspx, accessed on 5 September 2021)) and much more flexible for the storage of the metadata about attributes and operations whose number is highly variable moving from one class to another. From the previous two merits of the NoSQL solution, a third one follows, namely that the formulation of queries is easier than in the solution adopted in [28].

The Software Artifact Repository conceptual Model (briefly SARM) proposed in [35] is composed of three main concepts: SARM Content (in turn composed of Management Content, Model Content and Search Content), SARM Search Engine and SARM Interfaces (Figure 16). Our metadata repository implements the Search Content component, which, according to Hamid, must store metadata about the artifacts (e.g., UML class diagrams) in order to facilitate their location. Our solution delegates to the developed XMI parser the automatic extraction of information describing the artifacts in the model repository and their subsequent upload into the metadata repository. The Search Engine (Figure 16) manages the searching of the artifacts stored in the model repository. Hamid claims that three different modes of searching for artifacts are valuable: simple search, advanced search and browsing. Simple search offers support to general-purpose queries using keywords, while advanced search supports more complex queries. Search Interfaces use the Search Engine to search for artifacts in the repository.

Table 4. Comparison with the schema of the metadata repository in [28].

[28]	Our Repository
`class_Table`	`class`
`attributes_Table`	`attribute`
`operations_Table`	`operation`
`generalization_Table`	`parent` attribute of `class` table
`associationEnd_Table`	`associationEnd`
`association_Table`	`association`
`image_Table`	
`xmi_Table`	`packageID` attribute of `class` table
`realization_Table`	
`dependence_Table`	
	`project`
	`package`

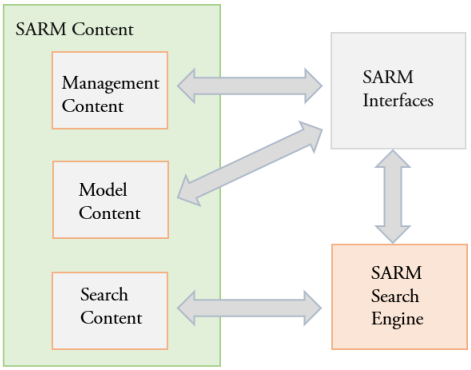


Figure 16. SARM architecture.

8. Conclusions and Future Work

The present study belongs to the domain concerning the automated feeding of repositories of UML diagrams, a topic that is considered relevant by most scholars. A software module was developed that is responsible for extracting useful information from the XMI file about class diagrams and entering it as metadata into a pre-existing (metadata) repository. The XMI parser has been implemented as a Java web interface. The structure of the metadata repository adopted in the paper comes from a UML metamodel and it has been implemented as a PostgreSQL database based on the JSONB data type.

The metadata repository together with the linked parser is a useful tool for companies operating in the market segment of advanced web applications. We believe that it can be of particular relevance for Small and Medium-Sized Enterprises (SMEs) developing software. This category of SMEs represents the majority of software development organizations around the world. "SMEs are made up of enterprises which employ fewer than 250 persons and which have an annual turnover not exceeding EUR 50 million, and/or an annual balance sheet total not exceeding EUR 43 million" (definition based on Article 2 of the Annex to Commission Recommendation 2003/361/EC; https://ec.europa.eu/docsroom/documents/42921, accessed on 23 September 2021). The development of web applications often requires a team of analysts of the "business model" and a team of programmers who implement what the first modeled. The number of human resources to be involved is, in general, conspicuous and with a high professional profile, factors that translate into high costs, which most SMEs are unable to afford. It follows that, to help SMEs to remain competitive or even to survive, a software development process that is not too complex, and easy-to-use tools for implementing it, are required. Lastly, the tools must be free of charge.

We understand that formulating NoSQL queries may be challenging. In order to make the proposed approach appealing, it is part of our future work to supplement the metadata repository with an additional web interface that implements advanced query mechanisms

for the retrieval of class diagrams in the model repository according to different criteria, as suggested, for example, in [15]. Such a web interface will implement the Search Interface discussed by Hamid in [35].

Author Contributions: Conceptualization, P.D.F. and G.P.; methodology, P.D.F.; software, R.P.; validation, M.M., R.P. and G.P.; writing, P.D.F.; funding acquisition, G.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by *Software Industriale*. https://www.softwareindustriale.it/en/gruppo-si-and-university.

Data Availability Statement: At link http://btc.digitalbusinessolution.com/menu/menuList/, the reader can find a simple Web application useful to check the results of the Case study. Vice versa, the scripts to create the Metadata Repository and the parser are available from the authors, since the xMR Parser is a proprietary software of Gruppo SI S.c.a.r.l. (https://www.softwareindustriale.it/en/gruppo-si-and-university).

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. The Schema of the Metadata Repository

The scripts for the creation of the tables composing the repository are shown below. The syntax is that of PostgreSQL 13.2 Documentation (https://www.postgresql.org/docs/13/, accessed on 2 September 2021).

```
CREATE TABLE project(
    projectID    serial          NOT NULL,
    name         varchar(255)    NOT NULL,
    PRIMARY KEY  (projectID)
);
CREATE TABLE package(
    packageID    serial          NOT NULL,
    name         varchar(255)    NOT NULL,
    URI          varchar(255)    NOT NULL,
    projectID    integer         NOT NULL,
    PRIMARY KEY  (packageID),
    FOREIGN KEY  (projectID)
    REFERENCES   project(projectID) ON UPDATE CASCADE
);
```

The URI provides a unique identifier for a package and remains unchanged once assigned.

```
CREATE TABLE association(
    associationID serial         NOT NULL,
    name          varchar(255)   NOT NULL,
    PRIMARY KEY   (associationID)
);
CREATE TABLE class(
    classID       serial         NOT NULL,
    name          varchar(255)   NOT NULL,
    packageID     integer        NOT NULL,
    associationClass boolean     NOT NULL,
    parent        integer,
    PRIMARY KEY   (classID),
    FOREIGN KEY   (packageID)
    REFERENCES    package(packageID) ON UPDATE CASCADE,
    FOREIGN KEY   (parent)
    REFERENCES    class(classID) ON UPDATE CASCADE
);
```

```
CREATE TABLE operation(
     classID       integer         NOT NULL,
     list          JSONB           NOT NULL,
     PRIMARY KEY   (classID),
     FOREIGN KEY   (classID)
     REFERENCES    class(classID) ON UPDATE CASCADE
);
CREATE TABLE attribute(
     classID       integer         NOT NULL,
     list          JSONB           NOT NULL,
     PRIMARY KEY   (classID),
     FOREIGN KEY   (classID)
     REFERENCES    class(classID) ON UPDATE CASCADE
);
CREATE TABLE associationEnd(
     associationEndID serial        NOT NULL,
     name             varchar(255)  NOT NULL,
     classID          integer       NOT NULL,
     associationID    integer       NOT NULL,
     PRIMARY KEY      (associationEndID),
     FOREIGN KEY      (classID)
     REFERENCES       class(classID) ON UPDATE CASCADE,
     FOREIGN KEY      (associationID)
     REFERENCES       association(associationID) ON UPDATE CASCADE
);
```

As stated in Section 4, the `Primary Key` constraint implements the OCL intra-UML-element constraint, while the `Foresign Key ... References` integrity constraint implements the relationship between classes. Notice that the latter constraint implements also the consistency of the values of attributes `parent` and `classID` of the `class` class.

Appendix B. The Instance of the Metadata Repository

The DataGrip (https://www.jetbrains.com/datagrip/) screen of Figure A1 lists the seven tables composing the metadata repository (Section 4); moreover, it shows an excerpt of the content of the database. In detail, the top screen shows the four tuples inserted into the `class` table (1 tuple for each class in the class diagram of Figure 15), the bottom-left screen shows the corresponding operations in those classes, while the bottom-right screen shows the association ends that describe the class diagram. In the diagram of Figure 15, neither the association ends nor the associations are explicitly named, so the parser adopts the OMG naming conventions of Section 3.

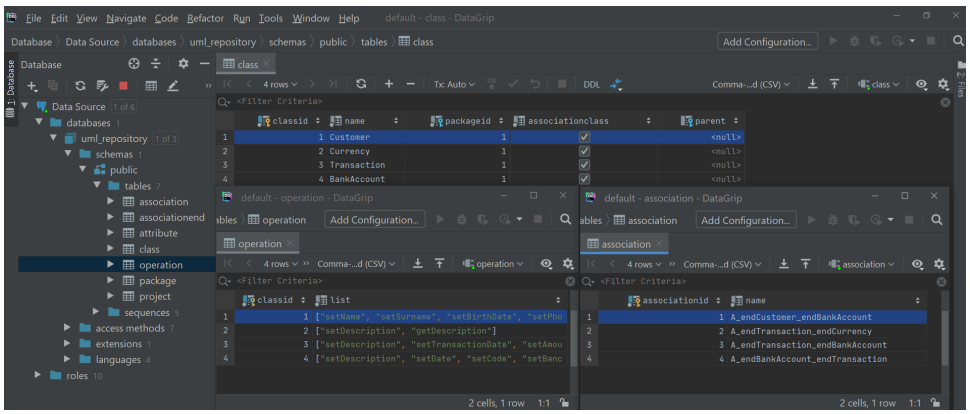


Figure A1. The tables of the metadata repository about class diagrams and a subset of their tuples coming from the case study.

In the absence of the developed parser (Section 5), the only method available for feeding the tables of the repository (with the metadata describing the class diagram of the case study (Figure 15)) consists in making recourse to SQL. For example, the statements that follow upload into the database, respectively, the operations and the attributes of the `Customer` class. The writing of these scripts is error-prone and time-consuming, so the benefits brought are obvious.

```
INSERT INTO operation(classID, list)
VALUES (1, '{
"Op1":  {"setName":  {"String":"name"}},
"Op2":  {"setSurname":  {"String":"surname"}},
"Op3":  {"setBirthDate":  {"Date":"date"}},
"Op4":  {"setPhoneNumber":  {"String":"phone"}},
"Op5":  {"setEmail":  {"String":"email"}},
"Op6":  {"getName":  "String"},
"Op7":  {"getSurname":  "String"},
"Op8":  {"getBirthDate":  "Date"},
"Op9":  {"getPhoneNumber":  "String"},
"Op10":  {"getEmail":  "String"} }');

INSERT INTO attribute (classID, list)
VALUES (1, '{
"Attr1":  {"name":  "String"},
"Attr2":  {"surname":  "String"},
"Attr3":  {"birthDate":  "Date"},
"Attr4":  {"phoneNumber":  "String"},
"Attr5":  {"email":  "String"} }');
```

Appendix C. Querying the Metadata Repository

The preliminary step towards the reuse of UML artifacts in the realization of new projects consists of investigating what is already available in the corporate class repository. In this direction, acquiring knowledge about the attributes and operations of the classes that are part of the UML class diagrams in the class repository is a mandatory step. This section lists a set of prototypical SQL queries against the metadata repository that can be used to reach such a goal.

Step 1: *Displaying of the contents of the* `project` *table.*
Query Q1 implements the request (Figure A2).

```
1  SELECT projectID, name
2  FROM   project;
```

Data Output | Explain | Messages | Notifications

	projectid [PK] integer	name character varying (255)
1	1	ATMProject

Figure A2. Query 1.

Step 2: *Displaying of the contents of the* `package` *table.*
Query Q2 implements the request (Figure A3).

```
SELECT packageID, name, URI, projectid
FROM   package;
```

Data Output | Explain | Messages | Notifications

	packageid [PK] integer	name character varying (255)	uri character varying (255)	projectid integer
1	1	myUMLClasses	C:\Users\utente\Desktop\myUMLRepository	1

Figure A3. Query 2.

Step 3: *Displaying of the names of the classes collected in a given package (packageID = 1) and the path of the latter.*

Query Q3 implements the request (Figure A4).

```
SELECT c.name, p.URI
FROM   class AS c, package AS p
WHERE  c.packageID = p.packageID AND p.packageID = 1
```

Data Output | Explain | Messages | Notifications

	name character varying (255)	uri character varying (255)
1	Customer	C:\Users\utente\Desktop\myUMLRepository
2	BankAccount	C:\Users\utente\Desktop\myUMLRepository
3	Transaction	C:\Users\utente\Desktop\myUMLRepository
4	Currency	C:\Users\utente\Desktop\myUMLRepository

Figure A4. Query 3.

Step 4: *Displaying of the attributes and operations in the* `customer` *class inside a given package (packageID = 1).*

Query Q4 implements the request (Figure A5). It is worth underlining the compactness of the JSONB format to visualize the organization of classes in terms of attributes and methods.

```
SELECT  jsonb_each(a.list) AS attributes,
        jsonb_each(o.list) AS operations
FROM    class AS c, package AS p, attribute AS a, operation AS o
WHERE   c.packageID = p.packageID AND p.packageID = 1  AND
        c.classID = a.classID AND c.classID = o.classID
```

Data Output | Explain | Messages | Notifications

	attributes record	operations record
1	(Attr1,"{""name"": ""String""}")	(Op1,"{""setName"": {""String"": ""name""}}")
2	(Attr2,"{""surname"": ""String""}")	(Op2,"{""setSurname"": {""String"": ""surname""}}")
3	(Attr3,"{""birthDate"": ""Date""}")	(Op3,"{""setBirthDate"": {""Date"": ""date""}}")
4	(Attr4,"{""phoneNumber"": ""String""...	(Op4,"{""setPhoneNumber"": {""String"": ""phone""}}")
5	(Attr5,"{""email"": ""String""}")	(Op5,"{""setEmail"": {""String"": ""email""}}")
6	[null]	(Op6,"{""getName"": ""String""}")
7	[null]	(Op7,"{""getSurname"": ""String""}")
8	[null]	(Op8,"{""getBirthDate"": ""Date""}")
9	[null]	(Op9,"{""getPhoneNumber"": ""String""}")
10	[null]	(Op10,"{""getEmail"": ""String""}")

Figure A5. Query 4. It uses the PostgreSQL's `jsonb_each()` operator.

A typical scenario is that in which the modeler queries the metadata repository in order to verify whether exist classes whose name looks like "customer". In the affirmative case, he is also interested in having an overview of the attributes and operations of these classes. Q5 implements such a search (Figure A6).

```
SELECT c.name, jsonb_each(a.list), jsonb_each(o.list)
FROM   package AS p, class AS c, attribute AS a, operation AS o
WHERE  p.packageID = c.packageID AND
       c.classID = a.classID  AND c.classID = o.classID AND
       ((c.name LIKE '%Customer%') OR (c.name LIKE '%customer%'))
```

Data Output | Explain | Messages | Notifications

	name character varying	jsonb_each record	jsonb_each record
1	Customer	(Attr1,"{""name"": ""String""}")	(Op1,"{""setName"": {""String"": ""name""}}")
2	Customer	(Attr2,"{""surname"": ""String""}")	(Op2,"{""setSurname"": {""String"": ""surname""}}")
3	Customer	(Attr3,"{""birthDate"": ""Date""}")	(Op3,"{""setBirthDate"": {""Date"": ""date""}}")
4	Customer	(Attr4,"{""phoneNumber"": ""String""}")	(Op4,"{""setPhoneNumber"": {""String"": ""phone""}}")
5	Customer	(Attr5,"{""email"": ""String""}")	(Op5,"{""setEmail"": {""String"": ""email""}}")
6	Customer	[null]	(Op6,"{""getName"": ""String""}")
7	Customer	[null]	(Op7,"{""getSurname"": ""String""}")
8	Customer	[null]	(Op8,"{""getBirthDate"": ""Date""}")
9	Customer	[null]	(Op9,"{""getPhoneNumber"": ""String""}")
10	Customer	[null]	(Op10,"{""getEmail"": ""String""}")

Figure A6. Query 5. It uses the PostgreSQL's `jsonb_each()` operator.

A not trivial inquiry concerns investigating the existence of hierarchies among tables in the class diagram of a given project. The complexity comes from the need to compute the transitive closure of all superclasses (if any) of a given instance of the `class` table (i.e., of a given instance of the UML element class of Figure 3). Recursive query Q6 implements such a request (Figure A7). As expected, there are no hierarchies among the classes of the class diagram of the case study.

```
WITH RECURSIVE RecQuery (classID, name, parent) AS
( SELECT  classID, name, parent
  FROM    class
  UNION ALL
  SELECT  child.classID, child.name, child.parent
  FROM    RecQuery AS parent, class AS child
  WHERE   parent.parent = child.classID )

SELECT DISTINCT classID, name, parent
FROM RecQuery
ORDER BY classID, name, parent;
```

Data Output | Explain | Messages | Notifications

	classid integer	name character varying (255)	parent integer
1	1	Customer	[null]
2	2	BankAccount	[null]
3	3	Transaction	[null]
4	4	Currency	[null]

Figure A7. Query 6.

References

1. Bucchiarone, A.; Cabot, J.; Paige, J.R.F.; Pierantonio, A. Grand challenges in model-driven engineering: An analysis of the state of the research. *Softw. Syst. Model.* **2020**, *19*, 5–13. [CrossRef]
2. OMG Unified Modeling Language, Version 2.5.1. OMG Document Number: Formal/2017-12-05 Normative. Available online: https/www.omg.org/spec/UML/ (accessed on 5 September 2021).
3. Rumbaugh, J.; Jacobson, I.; Booch, G. *The Unified Modeling Language Reference Manual*, 2nd ed.; Addison-Wesley: New York, NY, USA, 2005.
4. Di Rocco, J.; Di Ruscio, D.; Iovino, L.; Pierantonio, A. Collaborative repositories in model-driven engineering. *IEEE Softw.* **2015**, *32*, 28–34. [CrossRef]
5. Keller, R.K.; Bédard, J.-F.; Saint-Denis, G. Design and Implementation of an UML-Based Design Repository. In *Advanced Information Systems Engineering. CAiSE 2001*; Lecture Notes in Computer Science; Dittrich, K.R., Geppert, A., Norrie, M.C., Eds.; Springer: Berlin/Heidelberg, Germany, 2001; Volume 2068, pp. 448–464. [CrossRef]

6. France, R.; Bieman, J.; Cheng, B.H.C. Repository for Model Driven Development (ReMoDD). In *Models in Software Engineering. MODELS 2006*; Lecture Notes in Computer Science; Kuhne, T., Ed.; Springer: Berlin/Heidelberg, Germany, 2007; Volume 4364, pp. 311–317. [CrossRef]
7. France, R.B.; Bieman, J.M.; Mandalaparty, S.P.; Cheng, B.H.C.; Jensen, A. Repository for Model Driven Development (ReMoDD). In Proceedings of the 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012; pp. 1471–1472. [CrossRef]
8. Gosala, B.; Chowdhuri, S.R.; Singh, J.; Gupta, M.; Mishra, A. Automatic Classification of UML Class Diagrams Using Deep Learning Technique: Convolutional Neural Network. *Appl. Sci.* **2021**, *11*, 4267. [CrossRef]
9. Bernstein, P.A.; Dayal, U. An overview of repository technology. In Proceedings of the of the 20th International Conference on Very Large Data Bases, Santiago, Chile, 12–15 September 1994; pp. 705–713.
10. Mayr, C.; Zdun, U.; Dustdar, S. Reusable Architectural Decision Model for Model and Metadata Repositories. In *Formal Methods for Components and Objects*; Lecture Notes in Computer Science; de Boer, F.S., Bonsangue, M.M., Madelaine, E., Eds.; Springer: Berlin/Heidelberg, Germany, 2009; Volume 5751, pp. 1–20. [CrossRef]
11. Di Felice, P.; Paolone, G.; Paesani, R.; Marinelli, M. Overview of a Project devoted to Release an open-source Software Tool for the Creation, Feeding and Querying of a NoSQL Metadata Repository about UML Class Diagrams. In Proceedings of the 2nd International Electronic Conference on Applied Sciences, Basel, Switzerland, 15–31 October 2021. [CrossRef]
12. Postgres NoSQL: Combining Developer Productivity with Enterprise Data Integrity. An EnterpriseDB White Paper for DBAs, Developers & Database Architects, July 2014. Available online: info.enterprisedb.com/rs/enterprisedb/images/EDB_WhitePaper_Postgres_NoSQL.pdf (accessed on 10 November 2021).
13. Tran, N.V.; Ganser, A.; Lichter, H. Multi Back-Ends for a Model Library Abstraction Layer. In *Computational Science and Its Applications—ICCSA 2013*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2013; Volume 7973, pp. 160–174. [CrossRef]
14. Couto, R.; Ribeiro, A.N.; Campos, J.C. The Modelery: A Collaborative Web Based Repository. In *Computational Science and Its Applications—ICCSA 2014*; Lecture Notes in Computer Science, Part VI; Springer: Cham, Switzerland, 2014; Volume 8584, pp. 1–16. [CrossRef]
15. Basciani, F.; Di Rocco, J.; Ruscio, D.D.; Iovino, L.; Pierantonio, A. Model Repositories: Will They Become Reality? In Proceedings of the 3rd International Workshop on Model-Driven Engineering on and for the Cloud and 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015), Ottawa, QC, Canada, 29 September 2015.
16. Paolone, G.; Marinelli, M.; Paesani, R.; Felice, P.D. Automatic Code Generation of MVC Web Applications. *Computers* **2020**, *9*, 56. [CrossRef]
17. Paolone, G.; Clementini, E.; Liguori, G. A methodology for building enterprise Web 2.0 Applications. In Proceedings of the Modern Information Technology in the Innovation Processes of the Industrial Enterprises (MITIP), Prague, Czech Republic, 12–14 November 2008; pp. 228–233.
18. Génova, G.; Morillo, J.; Fraga, A. Metamodeling generalization and other directed relationships in UML. *Inf. Softw. Technol.* **2014**, *56*, 718–726. [CrossRef]
19. Queralt, A.; Teniente, E. Reasoning on UML Class Diagrams with OCL Constraints. In Proceedings of the 25th International Conference on Conceptual Modeling, Tucson, AZ, USA, 6–9 November 2006; pp. 497–512. [CrossRef]
20. Ritter, N.; Steiert, H.P. Enforcing Modeling Guidelines in an ORDBMS-based UML-Repository. In Proceedings of the 2000 Information Resource Management Association, International Conference on Challenges of Information Technology Management in the 21st Century, Anchorage, AK, USA, 21–24 May 2000; pp. 269–273.
21. Hebig, R.; Ho-Quang, T.; Robles, G.; Fernandez, M.A.; Chaudron, M.R.V. The Quest for Open Source Projects that Use UML: Mining GitHub. In Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems, Saint Malo, Brittany, France, 2–7 October 2016.
22. *OMG Object Constraint Language (OCL)*, Version 2.3.1. OMG Document Number: Formal/2012-01-01. Available online: https://www.omg.org/spec/OCL/2.3.1/PDF (accessed on 10 September 2021).
23. Makris, A.; Tserpes, K.; Spiliopoulos, G.; Zissis, D.; Anagnostopoulos, D. MongoDB Vs PostgreSQL: A comparative study on performance aspects. *Geoinformatica* **2021**, *25*, 243–268. [CrossRef]
24. Tortosa, A.H. Performance Benchmark PostgreSQL/Mongodb. A White Paper by Ongres. Available online: https://www.enterprisedb.com/white-papers (accessed on 10 October 2021).
25. Robles, G.; Ho-Quang, T.; Hebig, R.; Chaudron, M.R.V.; Fernández, M.A. An extensive dataset of UML models in GitHub. In Proceedings of the 14th International Conference on Mining Software Repositories, Buenos Aires, Argentina, 20–21 May 2017; pp. 519–522.
26. Karasneh, B.; Chaudron, M.R.V. Extracting UML Models from Images. In Proceedings of the 5th International Conference on Computer Science and Information Technology, Amman, Jordan, 27–28 March 2013; pp. 169–178. [CrossRef]
27. Karasneh, B.; Chaudron, M.R.V. Img2uml: A system for extracting uml models from images. In Proceedings of the 39th Euromicro Conference on Software Engineering and Advanced Applications, Santander, Spain, 4–6 September 2013; pp. 134–137.
28. Karasneh, B.; Chaudron, M.R.V. Online Img2UML Repository: An Online Repository for UML Models. In Proceedings of the 3rd International Workshop on Experiences and Empirical Studies in Software Modeling (Co-Located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems–MoDELS 2013), Miami, FL, USA, 1 October 2013.

29. Girgis, M.R.; Mahmoud, T.M.; Nour, R.R. UML class diagram metrics tool. In Proceedings of the International Conference on Computer Engineering & Systems, Cairo, Egypt, 14–16 December 2009; pp. 423–428. [CrossRef]
30. Gajewski, M.; Zabierowski, W. Analysis and Comparison of the Spring Framework and Play Framework Performance, Used to Create Web Applications in Java. In Proceedings of the IEEE 15-th International Conference on Perspective Technologies and Methods in MEMS Design (MEMSTECH), Polyana, Ukraine, 22–26 May 2019; pp. 170–173.
31. 2021 Java Developer Productivity Report. A Technical Report by JRebel, Perforce Software, Inc. 2021. https://mma.prnewswire.com/media/1422901/2021_java_developer_productivity_report.pdf?p=pdf (accessed on 8 July 2021).
32. Schlick, R.; Felderer, M.; Majzik, I.; Nardone, R.; Raschke, A.; Snook, C.; Vittorini, V. A Proposal of an Example and Experiments Repository to Foster Industrial Adoption of Formal Methods. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice. ISoLA*; Lecture Notes in Computer Science; Margaria, T., Steffen, B., Eds.; Springer: Cham, Switzerland, 2018; Volume 11247. [CrossRef]
33. Paolone, G.; Paesani, R.; Marinelli, M.; Di Felice, P. Empirical Assessment of the Quality of MVC Web Applications Returned by xGenerator. *Computers* **2021**, *10*, 20. [CrossRef]
34. Belaunde, M. A Pragmatic Approach for Building a User-Friendly and Flexible UML Model Repository. In *«UML»'99—The Unified Modeling Language. UML 1999*; Lecture Notes in Computer Science; France, R., Rumpe, B., Eds.; Springer: Berlin/Heidelberg, Germany, 1999; Volume 1723, pp. 188–203. [CrossRef]
35. Hamid, B. A Model Repository Description Language—MRDL. Software Reuse: Bridging with Social-Awareness. In *Software Reuse: Bridging with Social-Awareness. ICSR 2016*; Lecture Notes in Computer Science; Kapitsaki, G., Santana de Almeida, E., Eds.; Springer: Cham, Switzerland, 2016; Volume 9679, pp. 350–367. [CrossRef]