Bachelor Project

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of computers

Static Metamodel Generator for JOPA

Benjamin Rodr

Supervisor: Ing. Martin Ledvinka, Ph.D.
May 2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Rodr**	Jméno: **Benjamin**	Osobní číslo: **499086**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Generátor statického metamodelu pro knihovnu JOPA

Název bakalářské práce anglicky:

Static Metamodel Generator for JOPA

Pokyny pro vypracování:

1. Become familiar with the concepts of object model metamodel, static metamodel, annotation processing and Semantic Web basics.
2. Analyze the options for creating a static metamodel generator that would allow execution over uncompiled source code of a target application.
3. Design a static metamodel generator for the JOPA library that will allow execution during application build lifecycle in Maven.
4. Implement a static metamodel generator according to your design as an extension of the existing JOPA Maven plugin.
5. Evaluate your solution by comparing generated static metamodels of provided examples with manually created static metamodels.

Seznam doporučené literatury:

[1] M. Keith and M. Schincariol, Pro JPA 2: Mastering the Java™ Persistence API, Apress, 2009
[2] D. Allemang, J. Hendler, Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL, Morgan Kaufmann, 2011
[3] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, G. Bierman, The Java Language Specification, Oracle America, Inc., 2021

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Martin Ledvinka, Ph.D. skupina znalostních softwarových systémů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **26.01.2023** Termín odevzdání bakalářské práce: **26.05.2023**

Platnost zadání bakalářské práce: **22.09.2024**

Ing. Martin Ledvinka, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.

Datum převzetí zadání	Podpis studenta

Acknowledgements

I thank Mr. Ing. Martin Ledvinka, PhD. for the opportunity to create this thesis, and for his patience, assistance, and expert advice during the creation of this thesis.
I also thank my classmates, friends, and family for their support during the entire period of my undergraduate studies.

Declaration

I declare that I have prepared this thesis independently and that I have cited all the information sources used in accordance with methodological guidelines on ethical principles in the preparation of theses.

Abstract

The goal of this bachelor's thesis is to design and implement a static metamodel generator for the JOPA library. The thesis includes an analysis of current technologies in the Java language for working with code during compilation time, tools for generating static metamodels, and the selection of the best option. The selected option will then be implemented.

The outcome of this thesis will be a working implementation of the static metamodel generator for the JOPA library and a report about its functionality. This generator will be available in projects that are using the JOPA Maven plugin and will be executable via the terminal.

Keywords: Java, Static Metamodel, Generator, Compilation, Annotation processing, Semantic web

Supervisor: Ing. Martin Ledvinka, Ph.D.
E-305,
Karlovo nám. 13,
120 00 Praha 2

Abstrakt

Cílem této bakalářské práce je navrhnout a implementovat generátor statického metamodelu pro knihovnu JOPA. Součástí této práce je analýza nástrojů v jazyce Java pro čtení kódu během kompilace a nástrojů pro tvorbu statického metamodelu, a následný výběr nejvhodnější z možností. Na základě výběru této možnosti pak bude generátor vytvořen.

Výsledkem práce bude funkční implementace generátoru statického metamodelu pro knihovnu JOPA a zpráva o jeho funkčnosti. Tento generátor bude k dispozici projektům, které využívají JOPA Maven plugin, a bude spustitelný pomocí příkazové řádky.

Klíčová slova: Java, Statický Metamodel, Generator, Compilation, Annotation processing, Sémantický web

Překlad názvu: Generátor statického metamodelu pro knihovnu JOPA

Contents

Figures

Tables

Chapter 1
Introduction

The JOPA (Java OWL Persistence API) library [16] is a tool for working with semantic data in Java. One of its key features is the ability to map objects in Java applications to entities in an OWL ontology. JOPA enables developers to use, for example, the Criteria API, which provides a programmatic way to define and execute database queries. It is useful to have a static metamodel available to make working with the Criteria API easier and type-safe.

A static metamodel is defined in advance and can be used for query optimization, data validation, or code generation. It can be created manually, but generating it automatically would significantly simplify the work of a developer.

The compiler uses a static metamodel to check the validity of the queries, preventing runtime errors that might otherwise occur; therefore, the generator must access the code before compilation itself. The generator itself also has to be usable by developers, so it has to be integrated with building tools such as Maven or Gradle.

The main goals of this thesis are:

- To create a generator of static metamodel for JOPA-based object models.
- To allow integration of this generator into the build infrastructure of Java projects using the JOPA library.

The structure of this thesis is thus as follows:

Chapter 2 introduces the concepts of object model metamodel, static metamodel, annotation processing, Semantic Web basics, and others.

Chapter 3 analyzes the options for creating a static metamodel generator that would allow execution over the uncompiled source code of the target application.

Chapter 4 designs a static metamodel generator for the JOPA library according to the analysis and implements the static metamodel generator based on the proposed design as an extension to an existing Maven plugin for the JOPA.

Chapter 5 is about creating tests and evaluating the solution by comparing the generated static metamodel from the provided example with the static metamodel manually created by the thesis supervisor, Mr. Ing. Martin Ledvinka, PhD.

Chapter 2

Background

This chapter focuses on familiarizing the reader with the technologies and issues that my thesis addresses.

2.1 The Semantic Web

The internet is a source of a lot of information nowadays. However, it is understandable only by humans. For machines, it is difficult to read and work with. The semantic web [26] is the next evolutionary stage of the web. It is a concept that seeks to improve the Internet by adding meaning and structure to the data available there.

The Semantic Web consists of several different technologies and standards that together fulfill the purpose described in the previous paragraph. In this thesis, I will primarily describe RDF, OWL, and SPARQL.

2.1.1 RDF

RDF - Resource Description Framework [25] is one of the Semantic Web standards. It is a data representation language; therefore, it is quite abstract. There are different RDF serialization formats, such as RDF/XML, Turtle, and JSON-LD. The JSON-LD example is in snippet 2.1. The prefix "ex:" is a context shortcut, which simplifies access to resources used.

RDF uses a graph-structured data representation and allows for the definition of relationships between different resources. As the main identifier for resources, RDF uses URIs or IRIs; these global identifiers are one of its biggest features. Each resource is represented as a node in the graph, and relationships between resources are represented by an edge in the graph. For example, RDF can be used to represent the relationship from Figure 2.1 'Person can be a friend of a person' using nodes representing person Clark and Lois and an edge representing the relationship 'is a friend of'.

The language uses so-called triples to describe the data. A triple consists of a subject, a predicate, and an object. The subject is the resource to which the information in the triple refers; the predicate is the property that the triple describes about the resource; and the object is the value of that property.

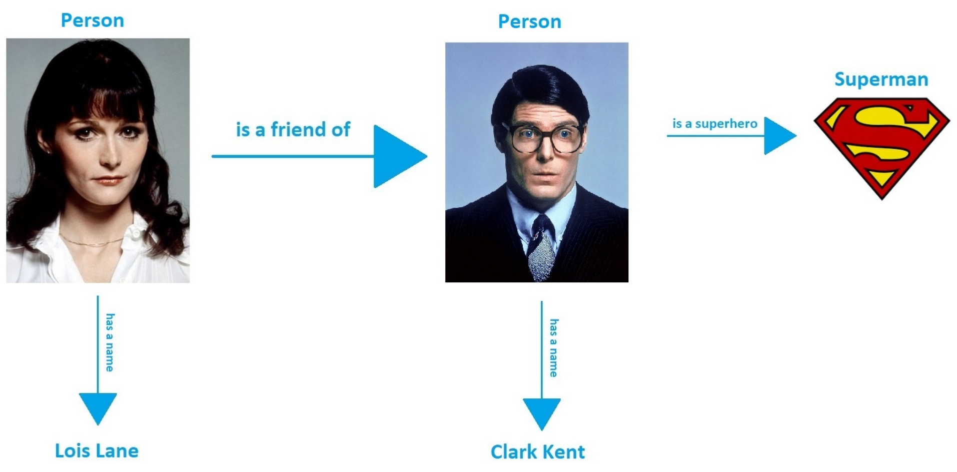


Figure 2.1: Example of an RDF graph

For example, the triple in Figure 2.1 'Clark is a Person' could be represented as '(Clark, is a, Person)'.

```
{
  "@context": {
    "ex": "http://example.com/"
  },
    "@id": "ex:Clark_Kent",
    "@type": "ex:Person",
    "ex:name": "Clark␣Kent",
    "ex:superhero":{
      "@id": "ex:Superman",
      "@type": "ex:Superhero",
      "ex:nickname": "Superman"
    },
    "ex:friend": {
      "@id": "ex:Lois_Lane",
      "@type": "ex:Person",
      "ex:name": "Lois␣Lane"
    }
}
```

Listing 2.1: JSON-LD example

RDF is flexible and can be used to describe different types of data and the relationships between them.

2.1.2 OWL

OWL - Web Ontology Language [24] is a modeling language and another standard of the Semantic Web. While RDF specifies the way data is serialized and structured, OWL as a modeling language gives meaning to the data, it brings instances and classes into the data.

OWL is used to define ontologies. An ontology is a formal model that describes the links between concepts and defines their properties and relations [22]. It helps computers understand different concepts. The role of ontologies is also to help with the disambiguation of bonds; for example, 'mouse' is ambiguous. It can be an animal mouse, however, it can also be a computer mouse. In these cases, where the bonds are very similar but still different, ontologies are very useful.

As said, RDF is used to describe resources and their relationships, and OWL is used to describe ontologies. For example, the 'SubClass' binding can be used to express the relation 'human is SubClassOf mammal', the 'equivalence' binding to express the relation 'Prime Minister of the UK and resident of 10 Downing Street are equivalent concepts', or the 'disjunction' binding to express the relation 'human is either male or female'.

There are currently two versions of OWL: OWL and OWL2. OWL2 is an extension of OWL and provides a larger set of relationships.

2.2 SPARQL

SPARQL - Simple Protocol and RDF Query Language [23] is a language for querying and manipulating data in RDF. SPARQL allows developers to create queries to retrieve and handle data from RDF graphs. Its syntax is similar to SQL and allows developers to create queries with conditions, aggregation functions, and sort operations. It also allows developers to use various operators to match and search expressions. However, SQL creates these queries by querying fields in tables. SPARQL does not use fields, but triple pattern matching.

SPARQL allows developers to retrieve data from various sources, such as web services, local files, or databases. However, a SPARQL endpoint is always required to execute the queries.

```
PREFIX ex: <http://example.com/>

SELECT ?hero
WHERE {
    ex:origin ex:krypton;
    ex:intentionOnEarth ex:helping.
}
```

Listing 2.2: SPARQL query

As mentioned above, the syntax is very similar to SQL, so there are also similar queries. Snippet 2.2 is a simple example of SPARQL. And 2.3 is then a similar query in SQL.

```
SELECT * FROM Hero
WHERE
    origin = 'krypton'
    intentionOnEarth = 'helping';
```

Listing 2.3: SQL query

Familiar keywords like 'SELECT' and 'WHERE' can be seen in the example; the '?' means variable, so the search is for a variable '?hero'. Next, there is an additional 'PREFIX' statement at the beginning of the snippet, and in the 'WHERE' statement, there is 'ex:origin ex:krypton'. This shows that the search is for someone who is not from Earth or another planet, but specifically from Krypton. The next statement, 'ex:intentionOnEarth ?ex:helping', is asking what the intention of a hero on Earth is. A hero's intention can be for example, helping, or getting rich, or it can be negative, such as conquering the world. So the whole query looks up all the heroes from Krypton who are on Earth to help.

2.3 JPA

JPA - Java Persistence API [18] is a standard Java interface for working with relational databases. It allows applications to create, store, modify, and retrieve objects from a database without any need of knowing the specific database system. Thanks to that, developers do not have to write SQL queries to communicate with the database and work with the results of the queries. They can instead work with objects that are directly mapped to tables in the database. This also allows developers to create applications without depending on a particular database system, and thus applications can be easily migrated to another database system. It is incorporated into Jakarta EE and builds on the JDBC standard implemented by databases such as MySQL, Oracle, PostgreSQL, and Microsoft SQL Server.

```
@Entity
public class Superhero {

    @Id
    private Integer id;

    private String nickname;

    @OneToMany
    private List<Superhero> associates;
}
```

Listing 2.4: JPA entity Superhero

```
public List<Superhero> findAll() {
    return em.createQuery("SELECT e FROM Superhero e",
      Superhero.class).getResultList();
}
```

Listing 2.5: JPA DAO example

JPA is designed to be as simple as possible and to allow developers to create applications with minimal effort. JPA also provides JPQL (Java Persistence

Query Language), which has syntax similar to SQL, but instead of working with tables, it works with entities and their attributes, so instead of searching for a match in a column, the developer can search for a match directly in the entity attribute. This allows developers to create applications faster and with less risk of producing errors. If you're more interested, Martin Fowler covers the topic of working with data in depth [15].

The snippet 2.4 shows the basic JPA entity class. In this class, it is shown that it has to have the '@Entity' annotation, which marks the class for the database. The next '@Id' annotation marks a field, which will be the identifier for this entity in the database table. When one of the fields is an array, it has to have an annotation, which defines what kind of relationship it has.

In snippet 2.5, there is a simple DAO (Data Access Object) class [3]. It uses Entity Manager, an interface used to interact with the entities and database [17]. The method 'findAll()' finds all entities in a table in the database.

2.4 JPA metamodel

A metamodel [20] is a model describing the structure and properties of another model. It is an abstract representation of a model that provides information about what kind of data the model can contain and what relationships can exist between these data.

JPA Metamodel exists since the creation of the persistence unit until its shutdown. It has several key elements that are important for its operation and use in applications. Some of which are:

- An entity type is a definition of a type, including classes, interfaces, enums, primitive types, and others. Every entity type contains its name, interface, references to other types, modifiers, annotations, and other properties. This information can be used for creating new code or validating existing code.
- Metamodel class, which allows developers to obtain information about classes and their properties, for example, attribute names or their data types. This information is used for creating queries and manipulating data. Thus, the metamodel serves as a layer of abstraction on top of the classes and allows developers to work with them at a higher level of abstraction.

As said, the metamodel is generated at the time of the creation of the persistence unit. The generator starts processing classes on the Java classpath. First, a few prerequisites are checked, such as whether the entity is valid, which means it has an identifier and an empty constructor exists. In the event that the supertype of the class is not processed, it will be processed first. After checking these prerequisites, processing of the fields begins. Fields are processed by their annotations. Lastly, named queries are processed.

There are two types of metamodels: dynamic and static.

A dynamic metamodel is used in JPA to map an object model to a relational database. The metamodel is generated at runtime based on entity classes using reflection.

The dynamic metamodel stores entity information such as table names, column names, and attribute types. This metamodel is then used to generate SQL queries and validate queries within the Criteria API. However, this is not the ideal way to use the Criteria API because it forces developers to define the query with string-based references, which are error-prone. Therefore, since the release of JPA 2.0, static metamodels are supported. It allows developers to use its fields as references.

2.5 Static metamodel

A static metamodel [20] is a set of classes that describe a model. It is available before the compilation and execution of code.

Snippet 2.4 is an example of a JPA entity for which a static metamodel is wanted to be generated; once again, this will be the Superhero class.

```
@StaticMetamodel(Superhero.class)
public class Superhero_{
  public static volatile Identificator<Superhero, Integer> id;
  public static volatile SingularAttribute<Superhero, String> nickname;
  public static volatile ListAttribute<Superhero, Superhero> associates;
}
```

Listing 2.6: Example of static metamodel class

In the snippet 2.6, there is the complete class of the static metamodel. The static metamodel class contains attributes that match the attributes of the entity in the model. The id attribute is of type Integer, the firstName, lastName, and nickname attributes are of type String, and the associates attribute is of type ListAttribute, which corresponds to the relationship between the superhero entity and other multiple superhero entities.

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Superhero> query = cb.createQuery(Superhero.class);
Root<Superhero> root = query.from(Superhero.class);

query.select(root)
      .where(
        cb.equal(root.get("nickname"), "Superman")
      );

List<Superhero> supermen = entityManager.createQuery(query).getResultList();
```

Listing 2.7: CriteriaAPI without a static metamodel

A static metamodel is useful, for example, in using CriteriaAPI [12]. Static metamodel attributes are used in queries instead of attribute names directly. For example, if a criteriaQuery is being created with a superhero entity class with a nickname attribute, we can use a static metamodel to access that attribute. Using the static metamodel attribute ensures that if a correct field is not referenced or the static metamodel is not successfully generated, the compilation will not succeed.

In the snippet 2.7, there is an example of simple CriteriaAPI usage. 'CriteriaBuilder' is created for creating queries. Then a query is created, and the root of the entity is created. The following work is straightforward.

The snippet 2.8 then shows the same query but with the use of the static metamodel. As can be seen, the string 'nickname' is replaced with a reference on static metamodel. This provides a safer search of the field.

```
{
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Superhero> query = cb.createQuery(Superhero.class);
Root<Superhero> root = query.from(Superhero.class);

query.select(root)
      .where(
        cb.equal(root.get(Superhero_.nickname), "Superman")
      );

List<Superhero> supermen = entityManager.createQuery(query).getResultList();
}
```

Listing 2.8: CriteriaAPI using a static metamodel

2.6 JOPA

JOPA - Java OWL Persistence API [27] is a library created by the CTU KBSS, aiming at efficient programmatic access to OWL2 ontologies and RDF graphs in Java. The system architecture and APIs are similar to those of JPA, making them familiar to developers who have worked with JPA.

The library allows developers to map objects to ontologies, use SPARQL queries to retrieve, create, edit, or delete data from ontologies, and integrate ontologies and their functionality into applications.

```
@OWLClass
public class Superhero implements Serializable {

    @Id
    private URI uri;

    @ParticipationConstraints(nonEmpty = true)
    @OWLDataProperty(iri = Vocabulary.p_nickname)
    private String nickname;

    @OWLObjectProperty(iri = Vocabulary.p_knows)
    private Set<Superhero> associates;

    @Types
    private Set<String> types;

    @Properties
    private Map<String, Set<String>> properties;
}
```

Listing 2.9: JOPA entity example

In snippet 2.9, an example of a JOPA entity can be seen. Right at the beginning, the annotation '@OWLClass' is used to indicate such an entity. The annotation '@Id' indicates, just as in JPA, that the field is an identifier. Unlike in JPA, this identifier must be of type URI or IRI, according to RDF.

Similarly, this example uses the annotations '@OWLDataProperty' and '@OWLObjectProperty', which mark the fields that are also meant to be persisted by the persistence layer. '@OWLObjectProperty' marks a reference type member, which means, for example, any type annotated with '@OWL-Class', whereas '@OWLDataProperty' marks a basic type member such as 'int', including also 'String' and 'Date'.

The '@Types' annotated field contains all OWL classes whose instance the particular individual represented by an object is (except the one mapped by the object's Java class).

'@OWLAnnotationProperty' marks fields that can be used to attach meta-data or annotations to various constructs in an ontology (not shown in the snippet).

Lastly, '@Properties' marks the field map that is not wanted to be mapped to the object model. For example, this allows developers to access properties that are new and not yet mapped.

```
private final EntityManager em;

public List<Superhero> findAll() {
    String soqlQuery = "SELECT h FROM Superhero h";
    em.createQuery(soqlQuery, Superhero.class).getResultList();
}
```

Listing 2.10: Persistent layer for Superhero class

The next snippet, 2.10, shows the persistent layer that will work with the superhero entity. In it, it can be seen that everything works very similarly to the way it works in JPA. There is a findAll() method that is using SOQL [28], it is an analogy to JPQL in snippet 2.5.It returns a list of all existing superheroes, exactly as in its analogous JPQL version.

2.7 Java Reflection API

The Java Reflection API [9] is a tool that allows browsing and retrieving information about classes, methods, constructors, and other elements of a Java program during runtime.

Using the Reflection API, a programmer can retrieve class information and dynamically manipulate that information. For example, he can get a list of methods that are available in the class and call the method using the information obtained. Other use cases may include creating test tools, program debugging tools, dependency injection frameworks, and more.

However, the Reflection API is intended for use with classes that are compiled. This means that code that is not compiled cannot be used with the Reflection API.

Snippet 2.11 shows a simple example. In this snippet, the class named 'Superhero' is found and its methods are retrieved. The name has to be a fully qualified reference, which means also including its package.

```
Class<?> superheroClass = Class.forName("cz.cvut.example.Superhero");
Method[] methods = superheroClass.getDeclaredMethods();
```

Listing 2.11: Simple Reflection API usage example

Chapter 3
Analysis

This chapter analyzes the problem and its possible solutions.

The goal is to create a static metamodel. This process can be divided into simpler, smaller problems. As shown in Figure 3.1, these smaller problems are class discovery, metamodel construction, and output file creation. Firstly, the classes wanted for the generator must be discovered. Then the metamodel must be created. Lastly, static metamodel classes must be generated from the metamodel itself.

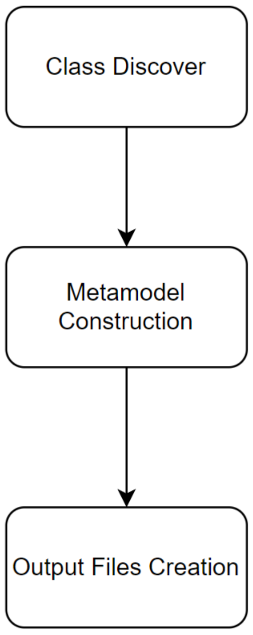


Figure 3.1: Process diagram of analysis

3.1 Existing solutions

As mentioned in the previous chapter, the static metamodel has many advantages in terms of development. It can simplify the developer's work, allowing him to validate his code, and generally speed up development.

Of course, static metamodel generators already exist. However, these are generators for the JPA standard, such as the Hibernate or EclipseLink libraries. They would not work in the JOPA library because of different data mapping terminologies and paradigms. Therefore, the goal of this work is to explore how a static metamodel can be created, select the optimal way, and then construct a working static metamodel generator.

3.2 Discovering classes to create a metamodel

The static metamodel is generated before compilation, which means that the generator will access and work with code that has not yet been compiled. From the code, it is necessary to get information about the classes from which the metamodel will be generated. In this case, these will be the classes annotated with '@OWLClass' and '@MappedSuperclass' which indicate that they are entities. This means that it will be necessary to somehow find and retrieve information about the classes that are annotated as such in the project. There are multiple ways of doing that; however, I was deciding between using the annotation processor or the ANTLR tool. The other option could be manual configuration, where the developer would provide a list of classes he wanted in a static metamodel, which would then be found. This option is very similar to using the annotation processor but more difficult because the annotation processor already does that based on annotations.

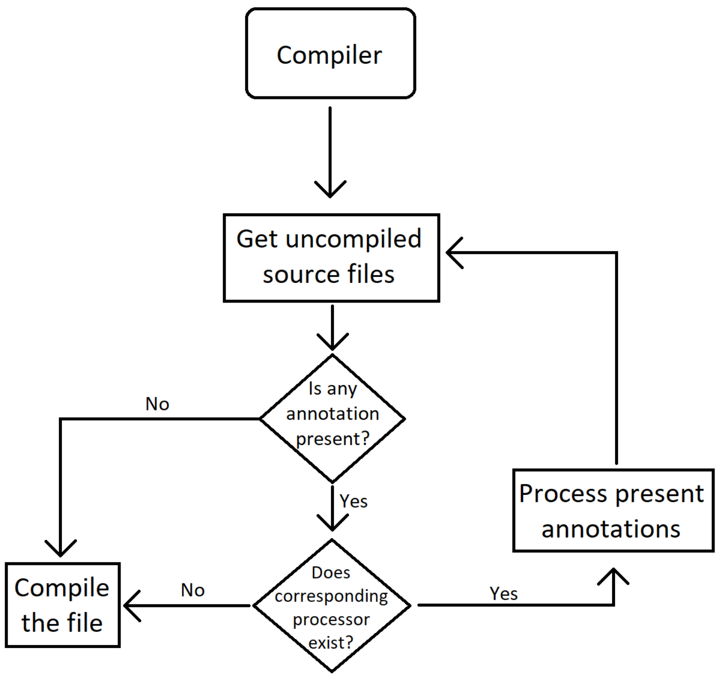
3.2.1 Annotation processing

Annotation processing [11] is a process that takes place right before compilation to parse and process the annotations that are used in the code. The 'javax.annotation.processing' library allows the creation of a program, an annotation processor, that executes this process at compile time.

This is a very powerful tool that allows extracting information about code and working with it during compilation.

The annotation processor then works as shown in Figure 3.2. During compilation, when the compiler finds an annotation, it tries to look for the corresponding annotation processor. In the event that it is found, the annotation process begins. The annotation processor then does its task, which is to generate new source files, validate existing code, or do something else. When the process is done, the source files are then added to the files that are being processed by the compiler. That means that if they contain annotations, the compiler will again try to invoke the processor, which is called the processing cycle [13]. Finally, the compilation outputs are created as class files that are executable by the JVM (Java Virtual Machine).

The processor has to eventually be used in the project that is supposed to be processed. This can be done in the case of Maven, for example, by using the maven-compiler-plugin and defining the annotation processor in the properties of this plugin.



Compiler
Get uncompiled source files
Is any annotation present?
No
Yes
Compile the file
No
Does corresponding processor exist?
Yes
Process present annotations

Figure 3.2: Diagram of annotation processing

Ultimately, annotation processors are very popular in static metamodel generators, so it would make sense to utilize them in this situation as well.

3.2.2 ANTLR

ANTLR - ANother Tool for Language Recognition [1] is a very widespread tool that helps with parser creation. A parser is a design pattern that takes text as input and transforms it into an organized structure that is readable by a computer [5].

As the definition of ANTLR implies, this tool can therefore be used to read source files as well. Thus, it can be used to create a parser that goes through all the source files of a project and then performs an operation on them.

In this case, the method of finding the wanted files would have to be defined. A disadvantage of this tool in this case is its inability to recognize programming languages. For this process, it would be useful if the tool could recognize which files are in fact Java files. For that, there would have to be another set of grammar rules defined with this tool.

3.2.3 ANTLR vs annotation processing

In the case of finding classes for the generator, the annotation processor and ANTLR can both be used, and both are valid options. Both technologies have their advantages and disadvantages. While the annotation processor is straightforward and can be integrated directly into a project, ANTLR requires writing a grammar and parser and is not as easy to use. On the other hand, ANTLR can be useful when more complex rules are needed to parse source code. However, in this case, it will only be necessary to locate classes by annotations and then process them into a metamodel. Therefore, creating a grammar and parser would be unnecessarily complex.

The choice of annotation processing could simplify the construction of the generator. A parser would be fairly time-consuming and would get us to the point where we are basically at the beginning of working with annotation processing. Also, as already mentioned, most of the generators use an annotation processor. Therefore, it is the obvious choice.

3.3 Metamodel construction

The idea is to use the metamodel generator, which is already integrated into JOPA. The method 'build()' is called on a 'MetamodelImpl' instance; it accepts a set of 'Class' instances as input to create the metamodel itself. That indicates that the method will use the Reflection API to obtain information about the specified class at runtime. The metamodel is then accessible through the 'getEntities()' method. It will return a set of 'EntityType' objects, which contain information about the metamodel classes.

3.4 Output files creation

After generating metamodel classes, the process of generating output files can begin. Again, as with obtaining classes, there are several ways of generating Java files. The first is to generate it using the custom output files generator. The second option is to use some kind of Java code generator.

3.4.1 Java code generator

Java code generator is a term for a tool for generating source code [8]. It allows a simple way of defining new source files, modifying them, and then generating them in defined directories. There are several Java code generators, but they all work very similarly and provide similar tools and results.

For this example, CodeModel was chosen. This tool is not very well documented; however, the blog [14] was very helpful in getting to understand what it can do. CodeModel has representations of Java classes, their fields, their methods, and other Java constructs. The snippet 3.1 shows the workflow. The developer can use CodeModel to create these constructs and then simply create output files. In it is shown the construction of the package, class, and its fields and methods.

```
JCodeModel codeModel = new JCodeModel();

JPackage definedPackage = codeModel._package("cz.cvut.example");
```

```
JDefinedClass definedClass = codeModel._class("Superhero");

definedClass.javadoc().add("Superhero class.");

JFieldVar nicknameField = jc.field(JMod.PRIVATE, String.class, "nickname");

JMethod getNickname = jc.method(JMod.PUBLIC, nicknameField.type(), "getNickname");

getNickname.body()._return(nicknameField);

codeModel.build(new File("."));
```

Listing 3.1: CodeModel usage example

In this example, a private field of type String 'nickname' is created. Then the method 'getNickname' is created, which will return the variable 'nickname'. Finally, the build method of the defined JCodeModel instance is called with the parameter 'newFile', which contains the path to the location where the source file should be stored.

This code will then generate a new source file, which will look exactly like the snippet3.2.

```
   package cz.cvut.example;


/**
 * Superhero class.
 *
 */
public class Superhero {

   private String nickname;

   public String getNickname() {
      return nickname;
   }

}
```

Listing 3.2: Result class generated by CodeModel

Of course, CodeModel offers more. For example, the generation of the constructor is not in this example; however, it is as simple as the generation of the methods that are in the example.

3.4.2 Custom output files generator

A custom output files generator means creating an output files generator exactly for the purposes of this static metamodel generator. That means after obtaining the metamodel, the generator will have to go through every EntityType object in the metamodel and create a proper source file for it. That will be done by using JavaIO's File and StringBuilder.

3.4.3 Custom output files generator vs CodeModel

As already mentioned with the custom output files generator, several problems will be encountered with its use that are not present with the use of CodeModel. CodeModel, despite its lack of documentation, would be easier to work with. Choosing a custom output files generator would lead to creating something that already exists, so the Codemodel is an ideal choice.

3.5 Execution

There are several options of execution for the static metamodel generator. It can be created as Java files, which are then runnable from the command line. It can be created as a standalone application that is runnable by itself. Or it can be created as a build tool plugin, which would then be implemented in the desired project.

The first option appears to be some sort of backlog. The second option appears good; however, the generator will work over some project, so a lot of work from the user's side would be required (for example, writing parameters for execution). As said, the generator will be used by the developer over some project, which makes the plugin an ideal choice. The developer would only need to set optional parameters because the plugin would already know what it needed to process.

Another advantage is that JOPA already has its own plugin, so implementation should not be particularly difficult.

Plugins are executed by the build tool, which is software that is used to automate the process of building Java applications and libraries [7]. This includes compiling source files, managing dependencies, packaging into JAR/WAR/EAR files, and more. The most popular of these tools are Apache Maven, Apache Ant, and Gradle [6]. The whole static metamodel generator should be integrated with such a build tool. In this case, since the whole JOPA library is using Maven, the selection of the build tool for the generator is straightforward, and therefore Maven will be selected. Another advantage of Maven is that it is compatible with Gradle, so the Gradle developers will also have a generator at their disposal.

As already mentioned, Maven is one of the most popular build tools. It uses an XML configuration file called 'pom.xml' for definitions of the project, its dependencies, profiles, plugins, and goals [2]. A plugin is a component that provides some functionality for a project. This plugin can be written either directly in Java or in other languages and can be used to perform various tasks within a project, such as compilation, packaging, testing, deployment, or a completely different custom task. A Maven goal is a concrete task that is defined in a plugin. For example, the maven-compiler-plugin provides a 'compile' goal that compiles source files. Each plugin can define different goals, and each goal will execute a different task. Goals can either be executed as single goals or they can be executed automatically as part of a particular phase of the project build, such as the 'compile' or 'package' phases. When a goal is executed, Maven looks up the corresponding plugin and executes the corresponding task. As said, JOPA already has its own plugin; therefore, the only thing remaining will be the creation of a goal for executing the generating process.

3.6 Reality check

During the programming of the final generator, several problems appeared. In this section, they will be addressed, described, and solved.

3.6.1 Metamodel construction problem

The annotation processing stage works as intended; however, a problem of incompatibility between annotation processor output and JOPA metamodel generator input appears. The annotation processor provides TypeElements and Elements. TypeElements are forms of annotations. Elements are the 'raw' form of classes and fields. The JOPA metamodel generator requires 'Class' instances because, as it is written in section 3.3, it uses the Reflection API. In that case, ANTLR may seem like a better option. Unfortunately, that is not true, because in both cases, the same problem will arise. The classes will not be compiled, and consequently, there will be a problem with the construction of the metamodel because they are not compatible with using the Java Reflection API.

Because of this problem, a new custom metamodel generator will be created only for the purposes of the static metamodel generator. The metamodel created by the generator will, however, not be a metamodel in terms of JOPA; it will only be a set of objects providing essential information for the creation of the static metamodel. A custom metamodel generator will take Element instances from the annotation processor, connect proper fields to proper classes, and make MetamodelClass objects ready for generating output files.

3.6.2 Java code generating problem

In snippet 3.1, it can be seen that CodeModel uses the Java Reflection API for creating parameters of methods and fields of classes, as already mentioned.

That means the problem of metamodel construction repeats itself in code generation.

One of the examples that shows this problem can be seen in the snippet 2.6. The static metamodel class has the annotation '@StaticMetamodel', which has the value 'Superhero.class'. This value is a reference to the Superhero class, which is not yet compiled. When using CodeModel or any other code generator, this annotation could not be created. The same problem appears with fields in this static metamodel class. In the same snippet, there are references to classes that are not yet compiled at the time of generation. Therefore, it means that its application is no longer a valid option, and so a custom output files generator must be created.

3.6.3 (Im)Possible workaround

The other possible solution to these problems could be to generate a static metamodel after compilation. However, that would mean an inability to use an annotation processor, which greatly simplifies the work. There could be alternatives to it; however, a different problem would appear.

As written in section 2.5, the main advantage of having a static metamodel available is easier and type-safe work with CriteriaAPI and descriptors. As seen in snippet 2.8, this approach also uses the Java Reflection API. Thus, the problem of an egg and a chicken would appear.

The CriteriaAPI could not be compiled because a static metamodel would not exist at the moment of compilation, and a static metamodel could not be created because compilation failed.

The workaround, which was in the beginning very attractive, is now impossible and would make problems even more difficult to solve.

Chapter 4
Implementation

The static metamodel generator will be programmed in Java, specifically in JDK version 1.8, due to compatibility requirements. As already mentioned, because of the usage of Maven in the whole JOPA library, the generator will also be using Maven as a build tool.

4.1 Static metamodel generator process

In this section, the whole process of generating a static metamodel will be explained. Figure 4.1 shows a sequence diagram of this process. As can be seen in the diagram, the process starts by calling the JOPA maven goal, which then creates and calls a task for the Java compiler containing the processor. The processor then generates a metamodel and calls OutputFilesGenerator to create a static metamodel.

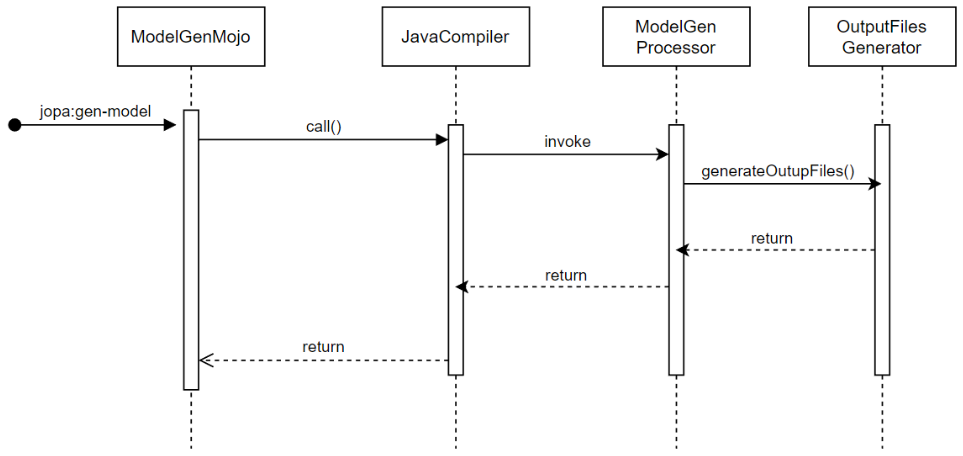


Figure 4.1: Sequence diagram of the static metamodel generator process

4.2 Project structure

The JOPA library is separated into individual modules, where each module fulfills its specific role. Static metamodel generator will be a new part of this library, and its structure will consist of a dedicated module for classes of the generator and a Maven goal class that will be integrated into the JOPA maven plugin module.

4.3 Generator module

As the name of this module implies, it contains all the logic of the static metamodel generator. The 'main' class of the generator is 'ModelGenProcessor'. It creates the metamodel and then calls 'OutputFilesGenerator' to invoke the process of generating final static metamodel classes with the given metamodel.

4.3.1 Class ModelGenProcessor

This class is the heart of the whole generator. It is invoked by the maven goal and calls all the methods.

The 4.1 snippet shows an example of what an annotation processor can look like [4]. Right at the beginning, there is an annotation '@SupportedAnnotationTypes', which specifies which annotations the processor should accept and therefore work with. It is a reference to a specific annotation class, and in this case, these are the '@OWLClass' and '@MappedSuperclass' annotations.

```
@SupportedAnnotationTypes({"cz.cvut.kbss.jopa.model.annotations.OWLClass",
  "cz.cvut.kbss.jopa.model.annotations.MappedSuperclass"})
public class ModelGenProcessor extends AbstractProcessor {

    @Override
    public void init(ProcessingEnvironment env) {
    }

    @Override
    public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment
      roundEnv) {
    }

}
```

Listing 4.1: Example of an annotation processor

Next, there is a new definition of the class 'ModelGenProcessor', and here it is visible that it inherits from the class 'AbstractProcessor'. This is a prerequisite for creating a custom annotation processor. This inheritance brings methods into the custom processor, some of which will be overridden.

The first method is 'init()'. This method accepts a ProcessingEnviroment as a parameter, which provides some useful tools, and it initializes the custom processor. This method is therefore something of a constructor.

The second method is 'process()'. This method is something of a main method for the custom processor. So, in this method, there will be concrete work with annotations. It accepts as parameters the TypeElemet set, which is the set of all the annotations that the processor will go through. RoundEnvironment is a variable that gives access to, for example, finding elements with a given annotation.

In this method, all the elements with wanted annotations will be found by browsing the TypeElement set and finding individual elements through RoundEnvironment.

4.3.2 Process of generating metamodel

For every element found during processing annotations, its information will be taken. From this information, the metamodel will be generated. The class diagram of the metamodel can be seen in Figure 4.2.

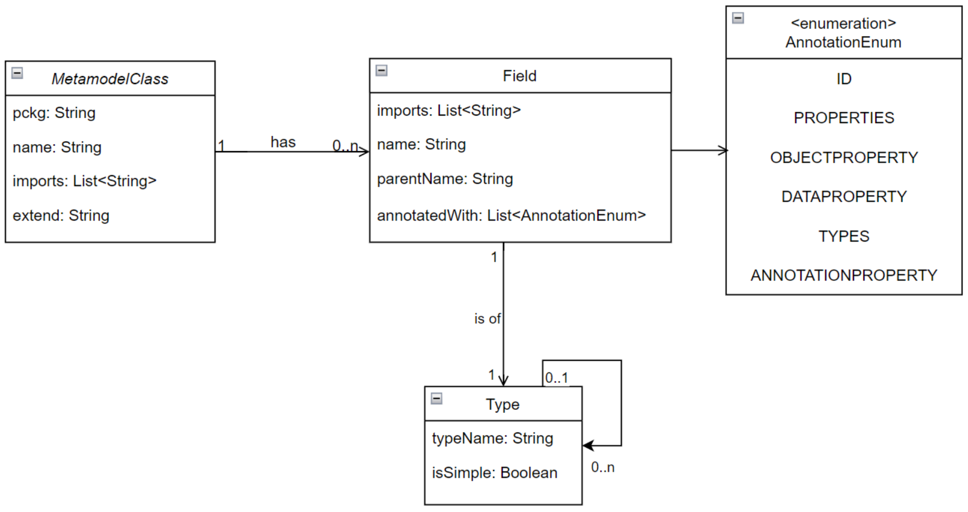


Figure 4.2: Metamodel class diagram

As seen on the class diagram, the whole metamodel consists of 'Metamodel-Class', 'Field', and 'Type', which are intertwined. 'MetamodelClass' consists of information about the class itself, for instance, name, package, or imports. 'Field' is a class containing information about distinct fields of classes, for example, name, parentName, or type. Type is information about the type of field. It contains the name, information about whether it is a simple field or not, and a list of types. This list is empty unless the type is not simple.

The metamodel does not contain methods of the class or fields that are not annotated with the OWL field annotations mentioned in 2.6.

4.3.3 Class OutputFilesGenerator

As the name suggests, this class controls the process of generating output files. When a metamodel is built, the annotation processor calls this class to start the generation process. This class first creates a file in the correct package, then starts processing its fields. Processing of fields is difficult because it depends on their annotation and type; for example, field 'id' of type integer annotated with '@Id' would, in the end, look exactly as in snippet 2.6. After fields are processed, the generator will finish the writing process, close the class, and save its text to a new '.java' file in the proper directory.

4.4 Class ModelGenMojo

This is a class that defines a new Maven goal. As said in the first two sections of this chapter. This class is an integration between the build tool (Maven in this case) and the annotation processor, which is the generator itself. It simplifies the creation of a static metamodel for the developer. The developer needs only to use the plugin, and a static metamodel will be created.

```
@Mojo(name = "gen-model")
public class ModelGenMojo extends AbstractMojo {

    private static final String OUTPUT_PARAM = "output-directory";

    @Parameter(alias = OUTPUT_PARAM, defaultValue =
      "./target/generated-sources/static-metamodel")
    private String outputDirectory;


    public void execute() throws MojoExecutionException {
    }
}
```

Listing 4.2: Example of a maven mojo class

In snippet 4.2 is shown the example and basic rules of Mojo class [21]. The class must inherit from the "AbstractMojo" class for it to work properly as a maven goal. This inheritance also provides the method 'execute()', which is the main method of the goal. Its code will be executed when the goal is called. That said, the class needs to be annotated with '@Mojo', which allows it to be called. This annotation has a parameter 'name', which serves to define the name of the goal. Last but not least, there is a field named 'outputDirectory', which gets the value of a parameter from properties in the plugin definition in 'pom.xml'. It means that when in plugin properties the 'output-directory' property is defined, its value will wire into the field 'outputDirectory'. If the value is not defined, the default value will be used instead.

Because the only task the goal has is to start the annotation processor, the compilation task must be invoked. For that, the object 'JavaCompiler' is used [19]. Through this object, the compilation task is created. The task has several input parameters. In this case, the most important parameters are 'compilationUnits' and 'options'. Parameter 'compilationUnits' is the set of units wanted to be compiled.

The parameter 'options' is a little bit more complex. It is a list of options for the compiler. There are a few prerequisites for the annotation processor to work. For it to be found, there needs to be a specific option in this list. It is option '-processor', followed by a reference to the processor class. Another prerequisite is the option '-cp'. This option defines the classpath for the compiler. The classpath is a list of directories and JAR files that the compiler uses to search for Java classes referenced in the classes being compiled.

When a Java program is being compiled, the compiler has to be able to find any external classes that are used by the program, such as third-party libraries or other classes in the project. The classpath tells the compiler where to look for these classes. That means that even the annotation processor will be in the classpath, and without it, the compiler will not know where to find the processor. Classpath files are found by retrieving them from 'MavenProject', a field retrieved by default in MavenMojo.

The next important option is '-d'. This option defines the destination directory where '.class' files will be generated by the compiler. In the event that this option is missing, these files will be generated at the source destination, which can then create confusion. Last but not least, these options also allow passing parameters to the annotation processor. The options that are supposed to be passed to the annotation processor need to be in the following format: '-A', followed by the name of the parameter, '=', and the value of the parameter. For example, if the parameter 'destination' with value './destination' needs to be passed to the annotation processor, the option would be '-Adestination=./destination'.

The Java compiler has several more options; however, these were the most important in this case.

After creating the task, it is executed and will call the annotation processor as expected.

4.5 Validation

The created static metamodel generator does not provide any validation for the generated static metamodel except for validation by compilation. The goal of this generator is to create a skeleton, not validate the skeleton. Thus, the correctness of the static metamodel will be found during the building of the dynamic metamodel. However, there are possible ways of validating it. For example, the developer can create tests specifically for his static metamodel, or several code analysis tools can be used for validation.

Chapter 5
Evaluation

This chapter is divided into two sections: the creation of tests for the generator and the testing of the generator on an existing large project.

5.1 Creation of tests for the generator

This section is about the creation of tests for the generator. Testing such a tool is not that simple. It can be divided into testing ModelGenMojo, ModelGenProcessor, and OutputFilesGenerator.

5.1.1 ModelGenMojo testing

Even though tests are really important for everything, the mojo only starts the annotation processor. It is simple and doesn't contain much logic, so the absence of tests is not critical.

5.1.2 ModelGenProcessor testing

Writing tests for the annotation processor is very difficult, and there are not many ways of doing it. The probably easiest is using Google's library 'compile-testing' [10]. This library is not exactly for annotation processor testing but rather compiler testing. However, it supports testing even with annotation processors, and thus it is a great suit for the purposes of testing ModelGenProcessor.

The library provides the method 'javac()'. This method returns the compiler builder, which then provides methods such as '.withProcessors()', '.withOptions()', and others. As their names suggest, the first one has an input parameter class implementing AnnotationProcessor. The second one provides options for the compiler. Using that parameter can provide, for example, the parameter 'outputDirecory' to the ModelGenProcessor.

After the set-up of the compiler is done, the builder method 'compile()' will initialize the compilation. This method has an input parameter of the library class JavaFileObject, which provides the ability to create a class for compilation from a string or file. The method 'compile()' returns a compilation object, which is then ready for validation. The 'compile-testing' library even provides its own assertion methods, which provide, for example, assertions about whether compilation succeeded without warnings or not. After that, assertions about whether the generated static metamodel class corresponds to the expected class follow.

The main purpose of ModelGenProcessor tests was to assert that the compilation finished successfully, that proper classes were used by the processor, and that for proper classes, proper fields were processed. Thus, two testing classes were created. These classes were "TestingClassOWL" and "TestingClassNotOWL"; as their names suggest, the first is a proper OWL class and the second is not an OWL class. Thus, tests assert that compilation finished without any errors and that only the first class was generated.

5.1.3 OutputFilesGenerator testing

For this class, JUnit tests were made. However, because the classes are very intertwined, testing individual methods is mostly not possible.

The tests were focused on asserting that the given field is annotated with the correct annotation, and that the generated file contains the proper fields. The tested methods were 'isAnnotatedWith()' and 'generateOutputFiles()'. The first method was easily tested by creating a field with an annotation and then asserting that the method returns a proper boolean. The second method was tested by creating a generated file before all tests and then asserting with each test that it contains proper fields. This method uses all the methods in OutputFilesGenerator, which means that in the event that any of these methods does not work properly, these tests will fail; however, the developer will not know which of the methods failed.

5.2 Testing on a large project

This section, as its name suggests, focuses on practical testing, where the generator will be tested on a large existing project.

5.2.1 Testing project

The project that was chosen for testing the static metamodel generator is TermIt. TermIt is a tool created by KBSS for terminology management based on the Semantic Web with a large object model containing over 40 classes annotated with '@OWLClass' or '@MappedSuperclass'. This project will therefore be an ideal test for the generator.

5.2.2 Method of testing

The plugin with the working goal will be applied to the testing project, and the goal will then be executed. The automatically generated static metamodel will then be validated by comparing its every individual class with the classes of the static metamodel that was provided by the supervisor of this thesis, Mr. Ing. Martin Ledvinka, PhD., and was created manually. In an ideal scenario, these two static metamodels will be identical, and no further repairs will be needed on the generator.

5.2.3 Results

This testing was definitely needed and successfully revealed flaws in the generator that were missed during development. Most of these flaws were not major, and one of the flaws was, for example, the incorrect configuration of the classpath parameter for the compiler in ModelGenMojo. Annotation '@Mojo' missed parameter 'requiresDependencyResolution', which defines what dependencies are needed for the goal; this meant that projects that used dependencies could not be processed by the generator and a static metamodel could not be generated. Another flaw was that the generator is supposed to ignore classes annotated with '@NonEntity'. The condition for this was then added to the code, and this scenario was then implemented into the ModelGenProcessor test. The last bigger flaw was that the generator is supposed to ignore fields annotated with '@Transient'. This condition was also added to the code, and a scenario was added to the ModelGenProcessor tests.

Chapter 6
Conclusion

The Semantic Web was supposed to be the future of the Internet. However, it is not the future of the Internet anymore. Its advantages are still forward-looking and can really change the view of the Internet for machines, but it is not possible to structure the whole Internet with the Semantic Web. JOPA is taking advantages of the Semantic Web and bringing them into information systems, where these advantages can be properly used.

This thesis explains what the Semantic Web is, what the metamodel and static metamodel are, and the importance of the metamodel and static metamodel in development.

It analyzes possible ways of creating a static metamodel generator and disassembles the large problem of creating the generator into smaller problems: class discovery, metamodel construction, and output files creation. It analyzes the difficulties that appeared while dealing with these problems and finds their solutions.

This work also described the concrete way of creating the generator by explaining the tools and technologies implemented in the generator. And last but not least, it tests and evaluates the functionality of this generator on a large project based on the JOPA library, where the results of the generator are compared to the static metamodel created manually by the supervisor of this thesis.

The goal of this thesis was to create a static metamodel generator for the JOPA library, and it has been successfully achieved. In the future, a few improvements could be integrated. Some of these improvements could be validation of generated metamodels or support of multiple inheritance in object models, which KBSS will implement in JOPA in the near future.

Bibliography

[1] ANTLR / Terence Parr. About The ANTLR Parser Generator, [accessed 2023-01-8]. https://www.antlr.org/about.html, 2015.

[2] Apache Software Foundation. Maven Getting Started Guide, [accessed 2022-11-30]. https://maven.apache.org/guides/getting-started/index.html.

[3] baeldung. The DAO Pattern in Java, [accessed 2023-01-13]. https://www.baeldung.com/java-dao-pattern.

[4] baeldung. Java Annotation Processing and Creating a Builder, [accessed 2023-04-08]. https://www.baeldung.com/java-annotation-processing-builder, 2022.

[5] Ben Lutkevich. What is a parser?, [accessed 2023-01-8]. https://www.techtarget.com/searchapparchitecture/definition/parser, 2022.

[6] devopscube. List of Open Source Java Build Tools, [accessed 2023-03-19]. https://devopscube.com/list-of-popular-open-source-java-build-tools/#:~:text=A%20java%20build%20tool%20is,%2C%20assembling%2C%20and%20deploying%20software.&text=Build%20tools%20are%20not%20only,and%20in%20continuous%20integration%20pipelines., 2022.

[7] Erica Vartanian. Java build tools: Maven vs Gradle, [accessed 2023-03-19]. https://www.educative.io/blog/java-build-tools-maven-vs-gradle, 2021.

[8] Gabriele Tomassetti. A Guide to Code Generation, [accessed 2023-03-19]. https://www.javacodegeeks.com/2018/05/a-guide-to-code-generation.html, 2018.

[9] Glen McCluskey. Using Java Reflection, [accessed 2023-03-19]. https://www.oracle.com/technical-resources/articles/java/javareflection.html#:~:text=Reflection%20is%20a%20feature%20in,its%20members%20and%20display%20them., 1998.

[10] Google. Compile Testing, [accessed 2023-04-25]. https://github.com/google/compile-testing, 2013.

[11] Hannes Dorfmann. Annotation Processing 101, [accessed 2023-01-8]. https://hannesdorfmann.com/annotation-processing/annotationprocessing101/, 2015.

[12] jboss. JPA Static Metamodel Generator, [accessed 2023-03-19]. https://docs.jboss.org/hibernate/orm/5.3/topical/html_single/metamodelgen/MetamodelGenerator.html.

[13] Karan Dhillon. All About Annotations and Annotation Processors, [accessed 2023-04-08]. https://medium.com/swlh/all-about-annotations-and-annotation-processors-4af47159f29d, 2021.

[14] Kevin Sookocheff. Generating Java with JCodeModel, [accessed 2023-01-8]. https://sookocheff.com/post/java/generating-java-with-jcodemodel/, 2016.

[15] Martin Fowler. Data Managemenet Guide, [accessed 2023-01-8]. https://martinfowler.com/data/, 2019.

[16] Martin Ledvinka, Petr Křemen. JOPA: Accessing Ontologies in an Object-oriented Way, [accessed 2022-10-20]. https://www.scitepress.org/PublicationsDetail.aspx?ID=p/CdcFwtlFM=&t=1, 2015.

[17] Oracle. Interface EntityManager, [accessed 2023-01-13]. https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html.

[18] Oracle. Introduction to the Java Persistence API, [accessed 2022-11-30]. https://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html, 2013.

[19] Oracle. JavaCompiler, [accessed 2023-04-08]. https://docs.oracle.com/javase/8/docs/api/javax/tools/JavaCompiler.html, 2014.

[20] Red Hat Software. Chapter 4. Metamodel, [accessed 2022-10-20]. https://docs.jboss.org/hibernate/stable/entitymanager/reference/en/html/metamodel.html, 2005.

[21] The Apache Software Foundation. Guide to Developing Java Plugins, [accessed 2023-04-08]. https://maven.apache.org/guides/plugin/guide-java-plugin-development.html.

[22] W3. VOCABULARIES, [accessed 2023-01-8]. https://www.w3.org/standards/semanticweb/ontology, 2015.

[23] W3C. SPARQL By Example, [accessed 2022-10-20]. https://www.w3.org/2009/Talks/0615-qbe/, 2008.

[24] W3C. OWL 2 Web Ontology Language Document Overview (Second Edition), [accessed 2022-10-20]. `https://www.w3.org/TR/owl2-overview/`, 2012.

[25] W3C. RDF 1.1 Primer, [accessed 2022-10-20]. `https://www.w3.org/TR/rdf11-primer/`, 2014.

[26] W3C. The Semantic Web, [accessed 2022-10-20]. `https://www.w3.org/standards/semanticweb/`, 2015.

[27] ČVUT KBSS. JOPA - Java OWL Persistence API, [accessed 2022-11-30]. `https://github.com/kbss-cvut/jopa#readme`.

[28] ČVUT KBSS. Semantic Object Query Language, [accessed 2023-04-08]. `https://github.com/kbss-cvut/jopa/wiki/Semantic-Object-Query-Language`.

Appendix A
Content of the electronic attachment

The electronic attachment to this work contains a text file named 'Static_Metamodel_Generator_for_JOPA_attachment.txt' which contains three internet links. The first [1] leads to my fork of the GitHub repository containing the created static metamodel generator; the second [2] and third [3] lead to pull requests of my code to the JOPA GitHub repository.

[1]https://github.com/syrcheddar/jopa
[2]https://github.com/kbss-cvut/jopa/pull/163
[3]https://github.com/kbss-cvut/jopa/pull/164