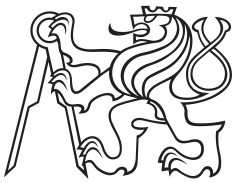
Master's Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Implementing Multiple Inheritance Support in JOPA

Bc. Jan Kolovecký

Supervisor: Ing. Martin Ledvinka, Ph.D.
Field of study: Open Informatics
Subfield: Software Engineering
May 2023

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Kolovecký Jan** Personal ID number: **475384**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Software Engineering**

II. Master’s thesis details

Master’s thesis title in English:

Implementing Multiple Inheritance Support in JOPA

Master’s thesis title in Czech:

Implementace podpory vícenásobné dědičnosti v knihovně JOPA

Guidelines:

1. Become familiar with Semantic Web languages RDFS and OWL (2) and their logic-based background. Familiarize yourself also with the persistence library JOPA.
2. Research and compare approaches of mapping multiple inheritance from RDFS/OWL (2) to Java.
3. Design an extension of JOPA that would allow the use of multiple inheritance in JOPA object models.
4. Implement support for multiple inheritance in JOPA according to your design.
5. Evaluate the correctness of your implementation by testing provided inheritance models. Ensure also backward compatibility of your changes to JOPA.

Bibliography / sources:

[1] M. Keith and M. Schincariol, Pro JPA 2: Mastering the Java™ Persistence API, Apress, 2009
[2] S. Harris and A. Seaborne, SPARQL 1.1 Query Language, W3C recommendation, W3C, 2013
[3] R. Cyganiak, D. Wood, and M. Lanthaler, RDF 1.1 Concepts andAbstract Syntax, W3C recommendation, W3C, 201

Name and workplace of master’s thesis supervisor:

Ing. Martin Ledvinka, Ph.D. Knowledge-based Software Systems FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **26.01.2023** Deadline for master's thesis submission: **26.05.2023**

Assignment valid until: **22.09.2024**

Ing. Martin Ledvinka, Ph.D. Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

Acknowledgements

I would like to thank Ing. Martin Ledvinka, Ph.D. for his continued support throughout both the bachelor's and master's theses. His dedication is only preceded by his attention to detail.

I also extend my thanks to my whole family, which has supported me endlessly through my studies.

My last thanks belongs to all of my colleagues, for I cannot imagine my studies without them.

Declaration

I declare that I have made the submitted work independently and that I have listed all the sources used in line with the Methodological Guideline on Compliance with Ethical Principles of preparation of academic final theses.

Prague, 22. May 2023

Jan Kolovecký

Abstract

JOPA is an established library for working with ontologies and semantic data in Java programming language. One of the features of semantic data is the ability of a class to have multiple parent classes. The aim of this thesis is to implement support for multiple inheritance in JOPA. This is complicated by the fact that Java does not support class-based multiple inheritance, therefore, a suitable method of emulating it must be chosen first.

The thesis consists of a description of relevant technologies, a discussion about ambiguities in multiple inheritance, a description and analysis of multiple methods of emulating multiple inheritance and of design and implementation of the selected solution. The last part evaluates the implementation by creating an application that takes advantage of multiple inheritance in JOPA.

Keywords: RDF, OWL, Ontology, Semantic technologies, JOPA, JPA, Java, Multiple inheritance, Diamond problem, OOM, OOP

Supervisor: Ing. Martin Ledvinka, Ph.D.

Abstrakt

JOPA je již zavedenou knihovnou zaměřenou na práci s ontologiemi a sémantickými daty v programovacím jazyce Java. Jednou z vlastností sémantických dat je schopnost třídy mít více tříd jako rodiče. Cílem této práce je implementovat podporu pro vícenásobnou dědičnost v knihovně JOPA. Toto je stíženo tím, že vícenásobná dědičnost není podporována v jazyce Java, tudíž nejdříve musí být vybrána vhodná metoda emulace vícenásobné dědičnosti.

Tato práce se skládá z popisu relevantních technologií, diskuze o nejednoznačnosti vícenásobné dědičnosti, popisu a analýze několika metod emulace vícenásobné dědičnosti a z návrhu a implementace vybraného řešení. V poslední části práce je provedená implementace zhodnocena vytvořením programu, který využívá vícenásobnou dědičnost v knihovně JOPA.

Klíčová slova: RDF, OWL, Ontologie, Sémantické technologie, JOPA, JPA, Java, Vícenásobná dědičnost, Diamantový problém, OOM, OOP

Překlad názvu: Implementace podpory vícenásobné dědičnosti v knihovně JOPA

Contents

Figures

Tables

Chapter 1

Introduction

At the turn of the century, the father of the World Wide Web, Tim Berners-Lee, envisioned the evolution of his invention. He created the World Wide Web as a system for information management. It was created for people to be used by people.

As such, it is hard for computer programs to understand and process the information on it. Even if a program can perfectly understand information on one page, this understanding does not translate to other pages, where the data structure is different, the same terms can have different meanings, and the author expects some prior knowledge from the reader.

Tim Berners-Lee, along with his colleagues, therefore proposed an extension to the World Wide Web, an information management system, which would be designed for machines. It was to be called the Semantic Web, a web where every piece of information is machine-readable and has a well-defined meaning. This would enable computer programs to make associations between pieces of information available and greatly enhance the ability to use the web for all machines [1].

Currently, the Semantic Web is a collection of standards defined by the W3C organization, and it uses many different technologies to support the idea. One of those technologies is the Web Ontology Language, a language used to represent the semantics of information on the web [2, 3].

Web Ontology Language (OWL) is an ontology language that enables users to build and use complex ontologies (descriptions of domain knowledge) [4]. These ontologies offer a unique view into a knowledge base and enable users to infer new knowledge and discover new relations and properties.

Developing applications that use these ontologies can be cumbersome and error-prone. For these reasons, the Java OWL Persistence API (JOPA) was created. JOPA enables developers to use simple abstracted API with Java classes instead of raw queries and statements. It also dynamically checks the data for integrity errors, stopping them from propagating further.

For developers, it is significant that the data models supported by JOPA and OWL models are philosophically as close to each other as possible. If there are significant differences between models, then the process from an idea to concrete action in ontology is much harder because the developer needs to be asking questions such as: “Is this action even possible in JOPA”

and “How does this JOPA action translate to change in OWL data”.

One of those differences that developers face today is multiple inheritance. Multiple inheritance is a feature supported and widely used in OWL and conceptual modeling in general but is not supported in JOPA. A subset of multiple inheritance features can be achieved right now, but its usage is both cumbersome and can lead to errors.

Therefore, this thesis aims to enable developers to use multiple inheritance in applications using the JOPA library in a way that is easy to use and can offer safety from errors caused by discrepancies between JOPA and OWL domain models. As the JOPA library is used in many projects, ensuring that all changes are backward compatible is vital.

In chapter 2, the relevant terms are defined together with a brief overview of JOPA. In chapter 3, the analysis of implementation is done, and these findings are used in chapter 4 to design the needed changes in the JOPA library. In chapter 5 the implementation of selected changes is described. The implementation is then evaluated in section 6. The thesis is then concluded in chapter 7.

Chapter 2
Background

This chapter provides the needed background for the terms, languages, and technologies used in this thesis.

2.1 RDF

RDF (Resource Description Framework) is simply a data model [5], which can flexibly express information about resources. RDF, as such, is abstract and therefore is not dependent on any domain or concrete implementation.

RDF data consists of triples (statements) with the following structure:

```
<subject> <predicate> <object>
```

A subject is a resource typically identified by the IRI. A resource is a concrete entity about which statements are being made.

The predicate describes relations between two resources, or a resource and literal[1]. IRI also identifies predicates. In fact, almost everything in RDF is identifiable by IRI. The predicates are directional. If the relationship is bidirectional, it needs to be explicitly stated.

An object is a resource with which a subject is in relation.

Statements can be visualized and used as directed graphs of knowledge [7]. Nodes are resources that are connected by properties. An example of such a graph can be seen in figure 2.1.

2.2 RDFS

RDF Schema (RDFS) is an extension of RDF that provides data modeling vocabulary. This vocabulary provides mechanisms for describing groups of related resources and relationships between resources [8].

RDFS enables the unified creation of hierarchies of classes and relations [9]. It also allows to specify domains and ranges of relations - limitations on which resources can be in the given relation.

[1] Literal is an atomic value - for example, a string or number [6]. As such, it is always an object in a triple, never a subject.

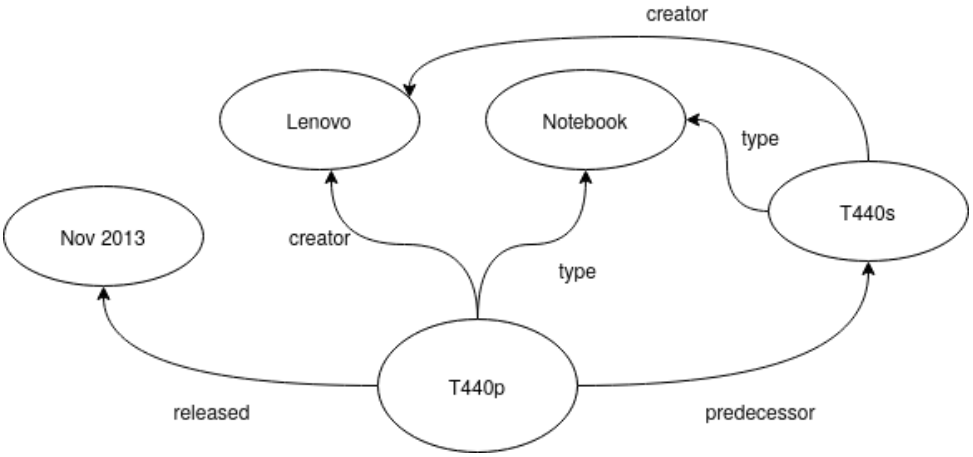


Figure 2.1: Example of RDF graph

The amount of metadata that RDFS brings to RDF can be seen in figure 2.2.

2.3 Ontology

The term ontology originates from the Latin word ontologia ("science of being"), and is used widely in philosophy, as a discipline studying reality, its nature, and structure [7, 9, 10].

When dealing with ontology in computer science, the term can be defined as a "formal, explicit specification of shared conceptualization"[11]. In other, more concrete words, an ontology in computer science is a description of some explicitly defined domain. Because the description is explicitly defined, it can be easily shared [12].

2.4 OWL

OWL 2 Web Ontology Language is a semantic web modeling language used for describing ontologies. It is more expressive than RDF(S), allowing a more precise description of a given domain [4].

The OWL describes relations between classes, instances, and properties in a declarative way. This information can be used in tools called reasoners to infer further knowledge from given data.

2.5 SPARQL

SPARQL Protocol and RDF Query Language (SPARQL) is a language for querying RDF datasets [13, 14].

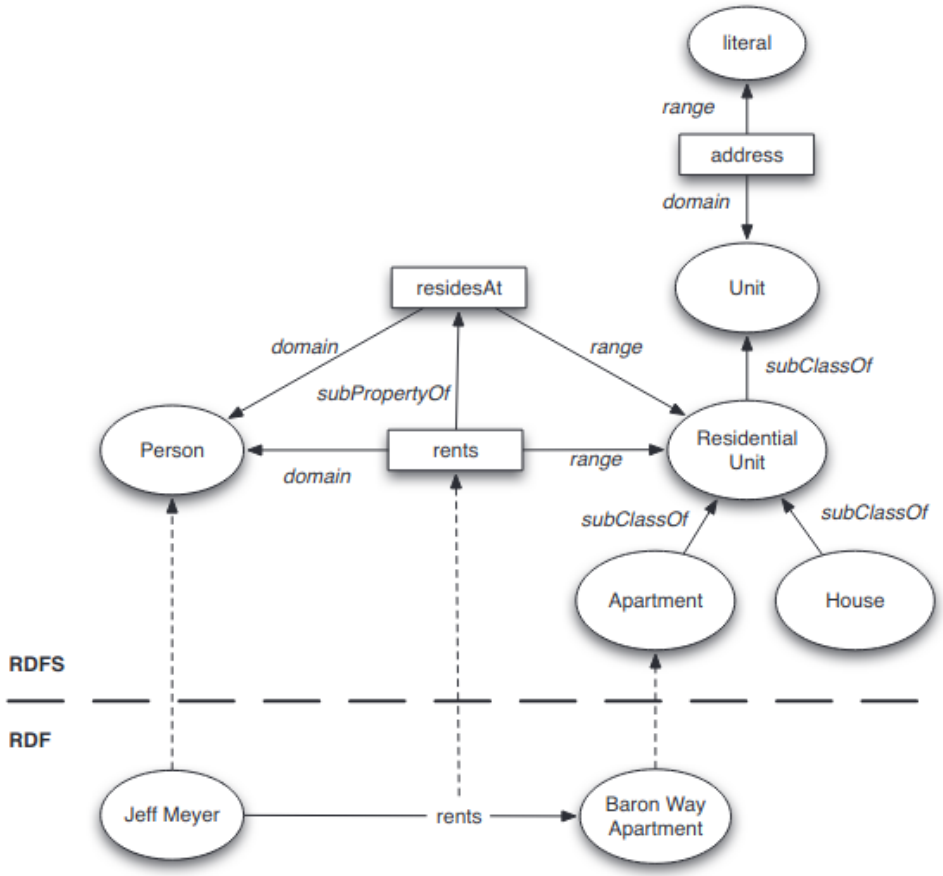


Figure 2.2: RDF and RDFS layers [6]

Its syntax bears some degree of likeness to the syntax of the SQL language, but its core principle is different. Whereas SQL is based on matching sets of single values, SPARQL is based on matching sets of triple patterns.

A simple example of a SPARQL query selecting all movies released in the year 2022, together with their titles, can be found in listing 2.1.

2.6 Object-relational and Object-ontological Mappings

Object-relational mapping (ORM) and object-ontological mapping (OOM) are two closely related terms. Both aim to ease working with data by mapping raw data in storage to objects and their attributes. This allows developers to treat data stored in a storage as if it were just simple objects in memory of the developed application, allowing developers to work with data more intuitively, and making it easier to write code that interacts with storage.

Object-relational mapping (ORM) maps raw data from relational databases to objects [15]. An example of a standard that defines ORM for the Java

```
PREFIX  sv:   <http://foo.bar/sv/elements/1.1/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

SELECT  ?movie ?title
WHERE
  { ?movie  sv:title     ?title ;
            sv:released  "2022"^^xsd:gYear .
  }
```

Listing 2.1: Example of SPARQL query

OWL term	JOPA term	JOPA Annotation
OWL Class	Entity class	@OWLClass
Instance (named individual)	Entity	-
Class assertions	Type specification	@Types
Property assertions	Properties specification	@Properties
Property assertion	Attribute	@OWLAnnotationProperty, @OWLObjectProperty, @OWLDataProperty

Table 2.1: Table showing JOPA terms along with their annotations, and their meaning in OWL namespace

language is JPA [16]. Should one venture into other programming languages, GORM[2] is an example of an OOM library for the Go language.

On the other side, object-ontological mapping maps raw data from ontologies to objects. Typical OOM libraries are JOPA[3] and AliBaba[4]

2.7 JOPA

Java OWL Persistence API (JOPA) is a Java OWL persistence library aimed at efficient programmatic access to OWL2 ontologies and RDF graphs in Java [17, 18].

To ease the adaptation of this library by developers, JOPA developers try to use the same mechanisms and terminology as JPA. For this reason, many terms used in this work are terms from the JPA specification [16].

Since OWL and JPA terminology differ, translations from OWL terminology to JOPA terminology, and the other way around, are necessary. Terms with corresponding meanings are displayed in table 2.1.

[2] `https://gorm.io` [visited on 22-12-16]
[3] `https://github.com/kbss-cvut/jopa` [visited on 22-12-18]
[4] `https://bitbucket.org/openrdf/alibaba/src/master/` [visited on 22-12-18]

2.7.1 Architecture

JOPA can be divided into two main parts.

- Ontodriver providing access to the actual storage. Deals only in axioms[5]. Its API is storage independent, with storage-dependent implementations.
- OOM realizes the object-ontological mapping, transactions, and meta-model functions.

The architecture of the JOPA library can be seen in figure 2.3. This figure shows the entire application stack, albeit simplified, from the user interface to storage.

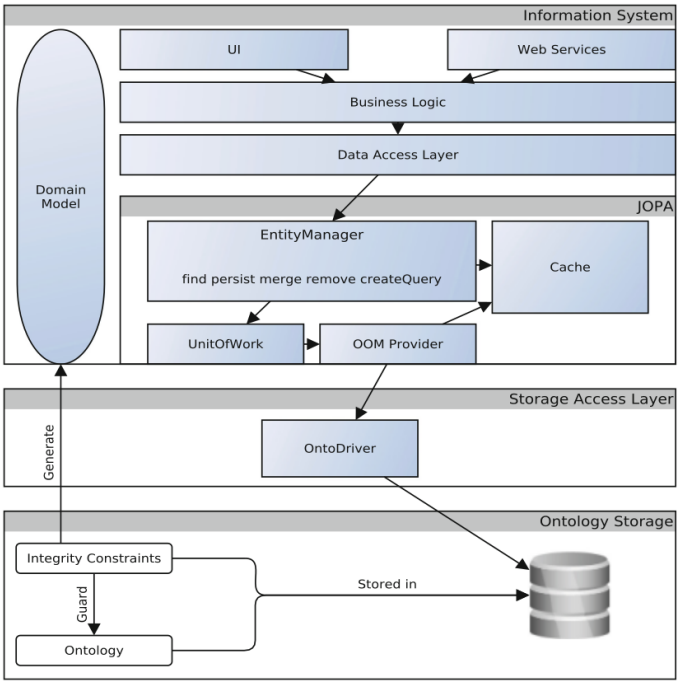


Figure 2.3: overview of JOPA architecture [18]

The part of JOPA with which its users interact primarily is represented by the `EntityManager` interface. Entity manager manages persistent context and can be used for basic CRUD operations, queries, and transactions. An implementation of the UnitOfWork design pattern represents persistent context, and it handles change management, (de)serialization of objects, caching and interaction with Ontodriver [19].

For creating instances, populating their attributes, and detecting associations JOPA needs extensive information about their structure, annotations, and more. This information could be extracted at runtime in each CRUD operation, but doing this would be highly inefficient.

[5] Axiom in the context of JOPA is either RDF triple or OWL axiom

For this reason, JOPA generates a metamodel at the start of the application. This metamodel has data about all model classes, their attributes, associations, and more.

2.7.2 Metamodel

As mentioned above, metamodel keeps data about entities throughout the application lifecycle. These data are generated at the start of the application.

The metamodel consists of the following key parts:

Entity type. The cornerstone of the metamodel is the entity type. This class keeps metadata about one particular model class and its attributes.

Whenever it is needed to serialize a class into axioms, the serializer needs to know which attributes are supposed to be serialized, which are not, their ontological type, the identifier of the class, and more.

The same applies to deserialization. During this process, the deserializer needs to know to which attribute which value belongs. The role of entity type can be seen in figure 2.4, where the simplified deserialization process is shown.

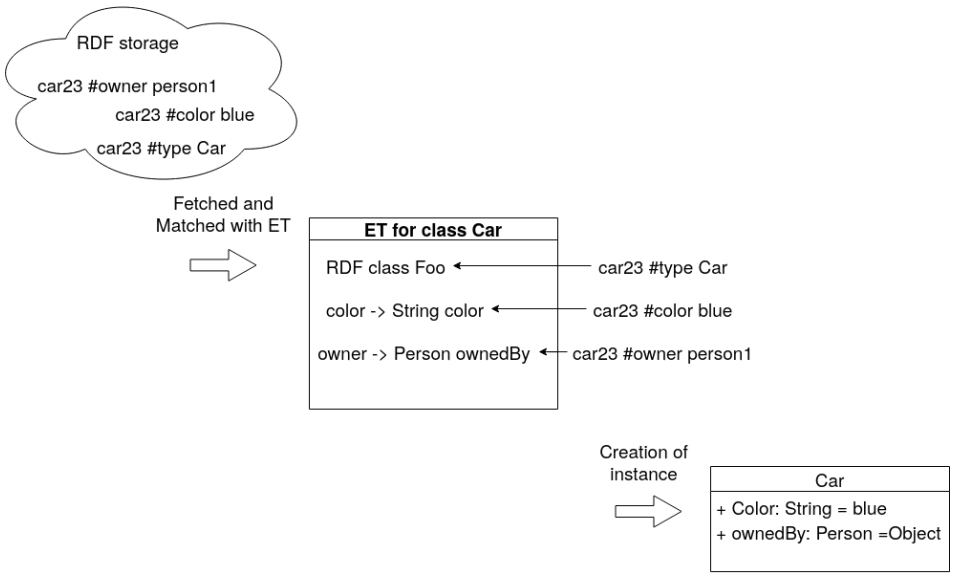


Figure 2.4: Simplified process of deserialization of RDF class

Metamodel class. The metamodel class keeps all entity types instances, along with the information about relations between entity types. Whenever an entity type has another entity type as a property, this relation is saved in the metamodel too.

Creation of Metamodel

The metamodel is created dynamically at the start of the application. At first, a class path scanner is run to detect all potential entity classes. These entity classes are determined by `OWLClass` annotation.

Then, all potential entity classes are processed. This consists of:

1. Checking all prerequisites are met. These prerequisites include the existence of no argument constructor and an identifier attribute.
2. Checking if the supertype of the class is already processed, processing it immediately otherwise.
3. Resolving lifecycle listeners[6].
4. Processing all fields. This does not include fields inherited from superclasses, as they are processed with their respective class.
5. Processing Named queries[7].

The most complex part of this process is processing the class fields because the field can belong to one of the multiple categories from the JOPA viewpoint. Mapping fields to their respective categories is done via property mapping annotations.

These categories are closely related to the OWL assertions from table 2.1, and are listed below, along with their respective property mapping annotations:

- OWL object property - `@OWLObjectProperty` - field whose value is another entity in OWL ontology.
- OWL data property - `@OWLDataProperty` - field whose value is literal in an OWL ontology.
- OWL annotation property - `@OWLAnnotationProperty` - field whose value is annotation property - commonly a description/commentary to OWL entity.
- Query attribute - `@Sparql` - field whose value is a result of some SPARQL query.
- Non-property attribute - `@Transient` - field not mapped in an ontology.
- Types - `@Types` - field containing all assertions of ontological classes of an instance.
- Properties field - `@Properties` - a map with unmapped attributes (property assertions) - assertions that are not defined in the model but exist in storage.

[6] Lifecycle listeners are methods that are defined by the developer and are called during object lifecycle events, such as before removing or after loading from storage.

[7] Named queries are a set of SPARQL queries that is coupled with entity class.

Fields from each of these categories need to be handled differently during (de)serialization, and even while accessing values stored in these fields. It is possible to define eager/lazy loading[8] on properties in JOPA, therefore JOPA needs to be able to load the value into the field when it is accessed.

Along with property mapping annotations, multiple other annotations can be declared on fields - annotations specifying participation constraints in relations, types converters, and more. These annotations are processed along the property mapping annotations.

2.8 Multiple Inheritance

"Multiple inheritance is good, but there is no good way to do it" [20]

Multiple inheritance in object-oriented programming (OOP) is the ability of class, or object, to inherit features from more than one parent. Multiple inheritance enables a greater degree of flexibility in a class hierarchy in comparison to single inheritance. However, with this flexibility comes several kinds of potential ambiguities in code, which cannot be easily dealt with. Therefore, many OOP languages support only single inheritance (class has at most one superclass), and may try to achieve some degree of the lost flexibility in other ways [21].

2.8.1 Diamond Problem

The diamond problem[9] is the most known ambiguity caused by multiple inheritance. This ambiguity occurs if a class has two ancestors who share a common predecessor. Then, from the point of the child class, there can exist multiple conflicting features in the hierarchy - different versions of methods with the same signature, states that can have different meanings, etc. [22, 23, 21].

One implementation-dependent characteristic of the diamond problem is the way the shared common ancestor is represented during program execution - whether there should be two distinct instances (replication), or one shared by all descendants (sharing).

The diamond problem can be looked upon from two points of view - ambiguity from a behavioral standpoint and from a state standpoint.

Behavioral Ambiguity

Behavioral ambiguity is ambiguity between methods inherited from parent classes. Figure 2.5 displays a simple example of behavioral ambiguity.

The class `Person` has method `payTaxes`, which is overridden in its descendant class `Student`. If method `payTaxes` is called on an instance of class

[8] When fetching an entity from storage, properties of this entity marked as eagerly loaded are fetched immediately. Lazily loaded properties are loaded dynamically by JOPA when they are being accessed for the first time [19].

[9] Also known as fork-join inheritance [22].

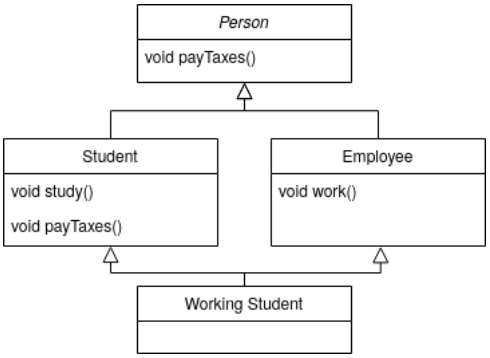


Figure 2.5: UML diagram of behavioral diamond problem

```
class D { int foo(){return 1;};};
class B : D {};
class C : D {int foo(){return 2;};};
class A : B,C {};

A a;

a.B::foo(); // 1
a.C::foo(); // 2

a.foo(); //Throws error, unless method foo was explicitly
  qualified in class A declaration by 'using C::foo';
```

Listing 2.2: Method qualification in C++. Same example hierarchy as in figure 2.5. Please note that parts of the code are omitted for brevity.

`Working Student`, should method `Person.payTaxes` or `Student.payTaxes` be called implicitly? Or both? If both, then in what order? Furthermore, if the method would return some value, how should the return values be combined?

Fortunately, this ambiguity is easily distinguishable by compilers (interpreters), and ambiguity can be resolved in child classes by overriding methods and explicitly qualifying method calls. Such a qualification can be used in the C++ language, and an example of such a qualification can be seen in listing 2.2.

This approach handles ambiguity but makes the hierarchy quite fragile toward changes in classes [21].

Although behavioral ambiguity is more severe if the common predecessor is shared, it can occur even with replication of the shared predecessor. The main ambiguity is in which order should the predecessors be methods called and in how to combine return values from classes.

This usually depends on the order in which inheritance is declared. Traversing the hierarchy may be needed in complex hierarchies to determine this order. For this, special algorithms are used, for example, the C3 linearization algorithm [24].

State Ambiguity

State ambiguity is not as easily dealt with as behavior ambiguity.

A model situation is displayed in figure 2.6. Here, `seniority` could stand for two different viable values, as an employee can have different `seniority` as a lecturer and as administrative staff.

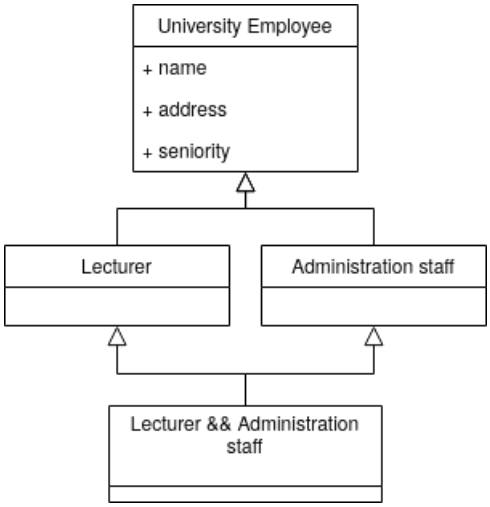


Figure 2.6: UML diagram of state diamond problem [22]

Having two distinct instances (replication) of `UniversityEmployee` in the hierarchy could solve this, but then a potential data inconsistency is created, with regards to `name` and `address`, as changes could be made to `UniversityEmployee` via the `Lecturer` instance, but they would not affect the `AdministrativeStaff` instance.

The trivial solution would be to move the `seniority` state to the lower level of the hierarchy. But that does not make sense from a modeling perspective, as all university employees share this property.

A better solution would exist if it were possible to choose if a class is shared or replicated in the hierarchy. Then the `UniversityEmployee` class could be split into two, one containing personal data (`name`, `address`) and the second one data on employment at the university. The first class, with personal data, would be shared, whereas the second one replicated [22]. This solution can be seen in figure 2.7. Setting class (inheritance) as either shared or replicated is possible in the C++ language by declaring inheritance as virtual [23, 25].

The example above is basic and relatively easy to solve with the right tools. However, more complex, and hard-to-solve situations can emerge from multiple inheritance. And dealing with these situations requires a rather significant change of hierarchy if done retrospectively, or excellent knowledge of the problem domain for designers to predict that such cases will occur in the future.

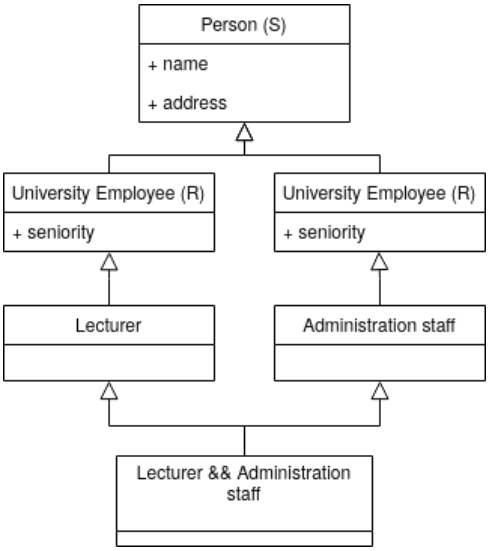


Figure 2.7: UML diagram of state diamond problem without ambiguity. S stands for shared, R for replicated.

2.8.2 Java and Multiple Inheritance

Java programming language does not support multiple inheritance. The main reason is that Java was created as a direct competitor to C++. During its development, James Gosling, the founder of the Java language, wanted to omit "many rarely used, poorly understood, confusing features of C++ that in our experience bring more grief than benefit". James Gosling directly denotes multiple inheritance as one of these confusing features [26, 27].

James Gosling also defined Java as a robust language. Robust meaning being reliable in a variety of ways, and multiple inheritance would jeopardize Javas' robustness.

Interfaces and Multiple Inheritance

Still, there exists some discussion about whether interfaces in Java should be thought of as a way to achieve some degree of multiple inheritance, as a class (or interface) can implement multiple interfaces [28, 29, 30, 31].

If one accepts that interfaces are Java's version of multiple inheritance, how does Java deal with the diamond problem?

State ambiguity. While it is possible to declare a field in an interface, the declared field will always be final and static. This renders interfaces capable of holding only a global, unmodifiable state. Furthermore, if a class or interface has multiple predecessor interfaces with two or more fields with the same name, it is required to specify the interface whose field is being accessed. Otherwise, an exception is thrown during compilation. This effectively eliminates state ambiguity [32].

```
interface D {
     default void foo() {System.out.println("D");}
     }

interface B  extends D{
    default void foo() { System.out.println("B");}
    }

interface C extends D{}

class A implements B,C {
    @Override
    public void foo() {
        B.super.foo(); /// prints "B"
    }
}
```

Listing 2.3: Method qualification in Java. Same example hierarchy as in figure 2.5.

Behavioral ambiguity. With version 8.0, Java added the possibility to declare default methods in interfaces. Default methods could theoretically become susceptible to the behavioral diamond problem. However, as stated in 2.8.1, qualifying method calls makes the behavioral diamond problem relatively easily solvable. Whenever a class or interface inherits two unrelated default methods that share the same arguments, name, and return type[10] Java compiler requires that the inheriting class or interface overrides this method [33]. An example of Java qualification can be seen in listing 2.3.

[10] More precisely, if two default methods are override-equivalent, see [33] for more details.

Chapter 3

Analysis

This chapter analyses the problem at hand. The main goal is to find the most suitable method to emulate multiple inheritance, first with general criteria, and then with regard to JOPA needs.

3.1 Implementing Multiple Inheritance

Even though an alternative version of the Java programming language could be created, and multiple inheritance implemented directly into it, this would not be the ideal approach, mainly because of its complexity and integration with other tools.

Fortunately, there exist a few methods that can sufficiently emulate multiple inheritance.

These methods will be introduced, demonstrated, and discussed in this section. The demonstration will be done by using the methods to implement a basic example, which can be seen in figure 3.1. Two base classes, `Notebook` and `Tablet`, with some methods and fields, are two parent classes of the class `Surface`. This hierarchy cannot be directly implemented in Java, as the `Surface` class extends two classes.

The discussion will focus on the advantages and disadvantages of methods and the differences between them.

At the end of this section, the most suitable method will be chosen to be used to emulate multiple inheritance in JOPA.

3.1.1 Interfaces with Default Methods

Even though Java class can extend (inherit from) only a single class, it can implement multiple interfaces. Since Java 8, there is also the possibility to implement default methods in interfaces [33]. These default implementations will be used if the descendent entity does not implement this function. Default methods have significant limitations, however. Because only final static fields can be declared in interfaces, only static final class fields are in scope in default methods. However, any non-static (and non-final) variable can be accessed through their respective setter and getter functions, given that they are declared in the interface. Therefore, it is possible to implement

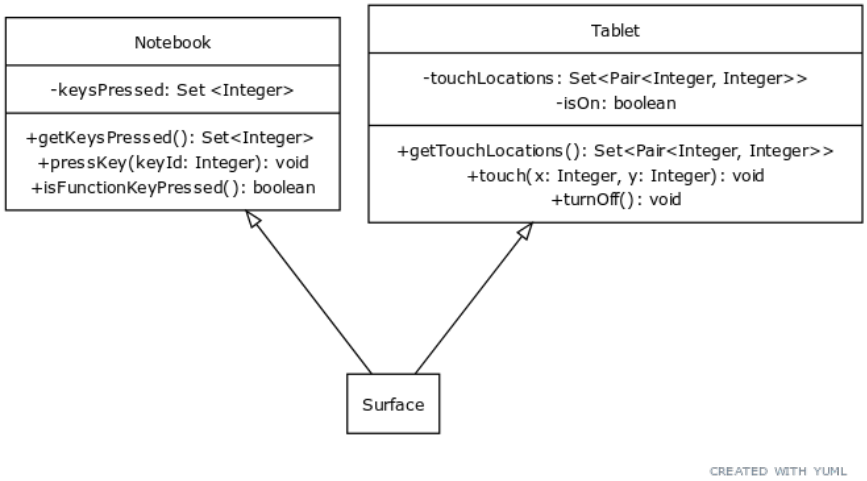


Figure 3.1: UML diagram of base example

almost anything that can be done in a standard class method by accessing all variables through getters and setters. In the descendent classes, only getters and setters need to be implemented, not all methods.

Implementing multiple inheritance using interfaces is relatively straightforward. For each would-be base class, an interface with the same methods as those in the base class is created. All methods, except getters and setters, are implemented as default methods. Then, a child class that implements all interfaces is created. In this class, all non-default methods from the interfaces are implemented, along with all the needed fields. See the listing 3.1 for a shortened example of the implementation of the base example, and figure 3.2 for the UML diagram describing this implementation.

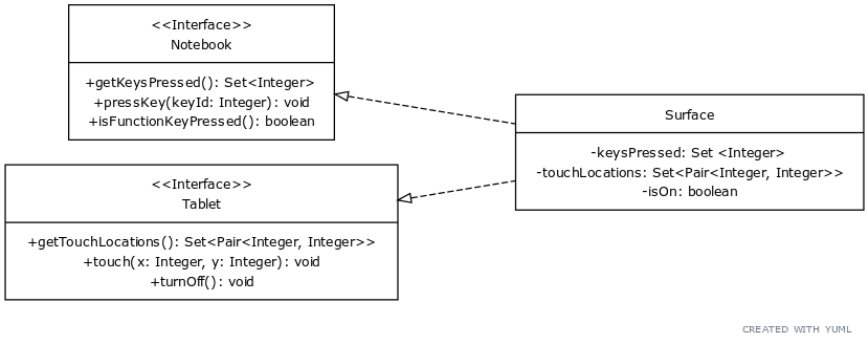


Figure 3.2: UML diagram of interface approach

As seen from the listings, the main disadvantage of this method is the need for re-implementing all non-default methods and all fields.

The main advantages are the simplicity of this method, its familiarity, and no need for external dependencies.

```
/// base interface
interface Notebook {
    Set<Integer> getKeysPressed();

    default void pressKey(Integer keyId) {
        getKeysPressed().add(keyId);
    }

    default boolean isFunctionKeyPressed() {
        return getKeysPressed().contains(42);
    }

}
/// child class - Tablet methods and fields omitted
class Surface implements Tablet, Notebook {
    Set<Integer> keysPressed = new TreeSet<>();

    @Override
    public Set<Integer> getKeysPressed() { /// only getter is
      implemented
        return keysPressed;
         }
}
```

Listing 3.1: Multiple inheritance via interfaces

3.1.2 Composition

Another approach is composition. In composition, the children's class consists of (composes of) multiple base classes. It is not as much inheritance as a delegation because the implementation of children's methods consists only of delegating the call to one of the base classes. The polymorphism is achieved using interfaces. The child class implements the same interfaces as base classes [34].

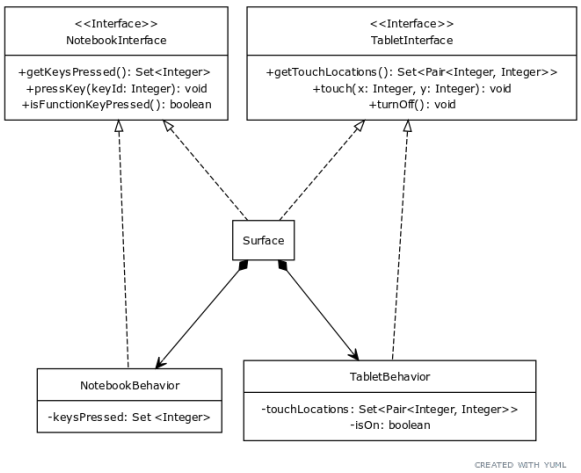


Figure 3.3: UML diagram of Composition approach

Similarly, to the interface method, for every base class, there exists a single interface. However, for every interface, a behavior class is created. This class implements the interface and all its methods and needed fields. Then, the

children class implements all interfaces and has all behavior classes as fields. Implementation of inherited methods consists of calling a given behavior class and returning the result.

Shortened implementation of this method can be seen in listing 3.2. UML diagram in figure 3.3 describes this concrete implementation.

This approach does have the advantage of not requiring to re-declare all class fields in every child class compared to the interface approach.

However, the disadvantages are clear - a lot of boilerplate code and the need to repeat the same lines of code in child classes.

3.1.3 Kivakit Mixins

Mixins can be thought of as an alternative/supplement to classic OOP. They are abstract classes that can be embedded into a class hierarchy to augment the capabilities of individual classes. Multiple mixins can be embedded into each class, adding new methods and fields.

Mixins may also be considered as a set of field and method declarations. They can add this set of fields and methods to any number of classes, and any class can have multiple independent sets [35]. This allows us to improve modularity and avoid code repetition.

Kivakit is a set of libraries/frameworks that aims "to provide a new vision for the development of Java software that takes code reuse to a new level." [36].

Among all the tools that should enable this goal is the KivaKit implementation of mixins. The implementation of multiple inheritance via KivaKit mixins can be seen in the listing 3.3

This implementation is more complex than the previous ones. All mixins have to be interface and extend concrete interface provided by the library. Because interfaces cannot have non-static non-final fields, every access to mixin properties must be via `Mixin.mixin()` function, which memorizes this value during execution.

This approach does not produce any boilerplate code and is simple to use.

Unfortunately, there are many disadvantages. One of them is the lack of a direct way to change values. There is only a getter for properties, so if a change in the value of the property is wanted, the property needs to be wrapped in some wrapper. Also, there is a limit of one property per mixin[1]. Another disadvantage of this approach is the way the mixin values are saved. They are all held in a global (static) map of all properties across all objects in JVM. This only brings about potential difficulties from memory and multi-threading perspectives.

[1] For more properties per one mixin the *AttributedMixin* interface must be used. This is a mixin that saves all values in a map. This effectively loses all information about the static typing of these properties.

```
interface Tablet {
    Set<Pair<Integer, Integer>> getTouchLocations();

    void touch(Integer x, Integer y);

    void turnOff();
}
/// behavior class - some methods omitted
class TabletBehavior implements Tablet {
    Set<Pair<Integer, Integer>> touchLocations = new HashSet<>();
    boolean isOn = false;

    public void touch(Integer x, Integer y) {
        if (!isOn) {
            return;
        }
        touchLocations.add(new Pair<>(x, y));
    }

    public void turnOff() {
        isOn = false;
        touchLocations.clear();
    }
}
/// child class - Tablet methods omitted
class Surface implements Notebook, Tablet {
    Notebook notebookBehavior = new NotebookBehavior();
    Tablet tabletBehavior = new TabletBehavior();

    @Override
    public Set<Pair<Integer, Integer>> getTouchLocations() {
        return tabletBehavior.getTouchLocations();
    }

    @Override
    public void touch(Integer x, Integer y) {
        tabletBehavior.touch(x, y);
    }

    @Override
    public void turnOff() {
        tabletBehavior.turnOff();
    }
}
```

Listing 3.2: Multiple inheritance via composition

```
/// child's class
class Surface implements Notebook, Tablet
{}

interface Notebook extends Mixin {
    /// returns saved value
    default Set<Integer> getKeysPressed() {
        return mixin(Notebook.class, HashSet::new);
    }

    default void pressKey(Integer keyId) {
        getKeysPressed().add(keyId);
    }

    default boolean isFunctionKeyPressed() {
        return getKeysPressed().contains(42);
    }
}
```

Listing 3.3: Multiple inheritance via KivaKit mixins

```
public aspect NotebookAspect {
    Set<Integer> Notebook.keysPressed = new TreeSet<>();

    public Set<Integer> Notebook.getKeysPressed() {
        return keysPressed;
    }

    public void Notebook.pressKey(Integer keyId) {
        keysPressed.add(keyId);
    }

    public boolean Notebook.isFunctionKeyPressed() {
        return keysPressed.contains(42);
    }

}
```

Listing 3.4: Aspect that modifies Notebook interface

3.1.4 AspectJ

AspectJ is a library that is used to implement the aspect-oriented programming paradigm in Java [37]. In addition to this, AspectJ allows developers to use extension methods. Extension methods allow developers to add methods, fields, and interfaces to existing classes and even interfaces. These modifications can be done through aspects - special constructs with Java-like syntax [37]. An example of such an aspect is shown in the listing 3.4. This aspect adds the `keysPressed` field and three methods to the `Notebook` interface, which can be seen with the child class in the listing 3.5.

AspectJ is a simple and quick approach, with only one significant disadvantage: the methods must be declared in aspect. For developers without knowledge of AspectJ, it can be dazzling where the methods are implemented, as they are not implemented in ordinary Java constructs. Another disadvantage could be the need for external dependency, but JOPA already uses

```
interface Notebook {
    Set<Integer> getKeysPressed();

    void pressKey(Integer keyId);

    boolean isFunctionKeyPressed();

}

class Surface implements Notebook, Tablet {
    // All methods are already implemented by
    // simply adding the interface!
}
```

Listing 3.5: AspectJ implementation of multiple inheritance

AspectJ for certain entity-related operations.

3.1.5 Javassist

Javassist is a Java library that enables the manipulation of Java bytecode, for example, modification and creation of new classes at runtime. This library can be used to implement multiple inheritance in a way very similar to the composition method, but without the need for boilerplate code. For this reason, the code snippet is not shown.

As stated above, this approach is very similar to composition, as both base interfaces and behavior classes implementing these interfaces are being used. However, a children class is not implemented manually, only an interface that extends all base interfaces. At runtime, the Javassist library creates a class that implements this interface and all inherited methods by delegating to concrete instances of behavior classes.

Compared to composition, the advantage is a reduction in both the boilerplate code and manual delegation in bodies of methods.

Unfortunately, there are many disadvantages, mainly the difficult, indirect creation of objects - the child's class must be instantiated in some factory method. This can cause difficulties in integration with different libraries used in the JOPA library. Also, bytecode manipulation is quite complex and can cause many unwanted behaviors.

3.1.6 Results

From the comparison of all the methods mentioned above, from the point of code reusability, boilerplate, and ease of usage, three methods were chosen for another comparison from the perspective of the JOPA library:

- Interfaces with default methods
- Composition
- AspectJ

3.1.7 Comparing with JOPA Needs

After choosing three promising methods from a general point of view, it is time to compare them with the JOPA library in mind.

Criteria

Backward compatibility. Because JOPA is not a new library, there exist many projects using it. Therefore, the chosen method must not require changes to already existing classes.

JOPA entity type class. The typical entity type class in JOPA is quite straightforward. A shortened example of such a class can be seen in listing 3.6. Several Java annotations can be seen on both the class and fields. The class has no methods with complex logic, only constructors, getters, and setters. The chosen method should reflect this fact.

Serialization and deserialization. The main part of JOPA is serializing and deserializing classes to and from axioms. The chosen method should use as much of the existing implementation as possible. The changes should follow the existing ideas.

Results

The results of the comparison of the chosen methods with the JOPA criteria are shown in table 3.1. Based exclusively on the table, the AspectJ method would seem like the most suitable method for implementing multiple inheritance. However, taking into account the comfort of users, namely the need for aspects in their source code, the interface method was chosen for implementation.

Also, at the time of writing, the decision to remove AspectJ dependency from JOPA was made, therefore effectively invalidating the AspectJ method.

3.1.8 Annotating Setters and Getters

The only significant disadvantage of the method chosen in the preceding section is the need for repetition of annotations on fields across the hierarchy. This could lead to potential inconsistencies where annotations and their values declared on the same OWL property could mistakenly differ.

Currently, JOPA annotations that specify attribute mapping (table 2.1) can be declared only on fields. Enabling these annotations on getters and setters would enable developers to define the attributes and their values in parents, leading to a single source of truth.

This is also supported by JPA and its annotations, where the annotation on getters and setters defines whether the field should be accessed directly or by its accessor methods[2] [16]. Even though JOPA aims to use the same

[2] The term accessor (method) means a method that is either a getter or a setter to some field of a class [38].

	Backward compatibility
Interfaces	No changes needed
Composition	No changes needed
AspectJ	No changes needed
	JOPA entity type class
Interfaces	Fields, together with annotations on them, must be declared in child classes, this could lead to inconsistencies across hierarchy
Composition	Supports all traits
AspectJ	Supports all traits
	Serialization and deserialization
Interfaces	Minimal changes needed
Composition	Significant changes needed
AspectJ	Minimal changes needed

Table 3.1: Table of comparisons of different methods of implementing multiple inheritance with JOPA needs

mechanisms as JPA, and this thesis should aim to uphold this, it will not be in the scope of this thesis to implement the support for specifying access to fields in JOPA. This thesis will use the annotations on accessors only as a way to create a single source of truth.

```
@OWLClass(iri = Vocabulary.notebook)
public class Notebook {

    @Id
    private URI id;

    @ParticipationConstraints(nonEmpty = true)
    @OWLDataProperty(iri = Vocabulary.created_on)
    private Instant created;

    @ParticipationConstraints(nonEmpty = true)
    @OWLObjectProperty(iri = Vocabulary.owned_by)
    private URI owner;

    @OWLObjectProperty(iri = Vocabulary.pressed_key)
    private Set<Key> keysPressed;

    public Notebook() {
    }

    /// other constructs ommited

    public Instant getCreated() {
        return created;
    }

    public void setCreated(Instant created) {
        this.created = created;
    }
    /// other getters and setters ommited
}
```

Listing 3.6: Snippet showing JOPA Entity Type class and its annotations. Certain methods were omitted for brevity.

Chapter 4

Design

As JOPA is a complex library, changes must be designed carefully so they can be integrated with already existing code, and do not break already existing functionality.

In this chapter, the parts of JOPA which could need some augmentations are identified, discussed, and if any augmentation is needed, it is designed.

This chapter is divided into two parts. In the first part are discussed and designed changes needed for support of multiple inheritance, and in the second, changes needed to support annotations on methods.

4.1 Multiple Inheritance

The metamodel module is the main part of JOPA that needs to be changed. The reason is that it reflects the hierarchy of ontology classes in itself. This module needs to reflect all new possible class hierarchies that come with multiple inheritance.

Only a few, if any, changes are expected in the other parts of the library.

4.1.1 Abstract Identifiable Type

As mentioned in analysis (2.7.2), the entity type contains metadata about a single entity class. However, the `EntityType` is a simple interface in JOPA that is implemented by the `AbstractIdentifiableType` (AIT) class.

As such, AIT keeps a number of metadata about an entity class. One is a `supertype` field whose value is another AIT class (if a class has a parent). This field must be changed to accommodate the possibility of multiple parents.

The primary reason for AIT to have this kind of a reference is that if the entity class extends some parent class, it inherits all attributes of that parent class. Information about these attributes is not duplicated in the child class, and so the information is queried from the parent(s) recursively at runtime. AIT checks its attributes, if the queried attribute is not found among them, the instance queries its ancestors recursively, and either the attribute is found, or the root of the hierarchy is reached.

After the changes to the supertype field, there is no need to change the attribute querying method drastically, as attributes can be declared only in

class parents, not interface parents. And since Java class can have only one parent class, only one parent will be queried. AIT will need to find this class parent among all other parents before recursively querying. This is needed because it is vital to remember all parents, including interface parents, as they are needed for other reasons, such as resolving lifecycle listeners.

4.1.2 Metamodel Creation

Other changes are needed while creating a metamodel.

When scanning classes for manageable classes, a class must satisfy two conditions to be considered manageable. It must have an `@OWLClass` annotation and must not be an interface. This must be changed to allow parent interfaces to be processed.

In addition, AIT has the `subtypes` field, which is used in deserialization. Implementation needs to guarantee the addition of all children's classes to their respective superclasses during metamodel creation and vice versa.

4.1.3 Deserialization

When JOPA tries to deserialize some abstract class, it must determine to which nonabstract entity it should deserialize. This is done in the class `PolymorphicEntityTypeResolver`, where the hierarchy is recursively traversed to find the most specific non-abstract type.

This procedure could be reused when JOPA needs to deserialize some parent entity type, which is an interface to find its descendant, which can then be instantiated.

The decision of whether to find a nonabstract descendant or to deserialize the class directly depends on the class being abstract from the point of view of JOPA, more concretely on method `AbstractIdentifiableType.isAbstract`. This method returns `true` if the type is abstract, and `false` if otherwise.

This method has to be modified to include interface types as abstract types.

4.2 Method Annotating

As proposed in 3.1.8, annotations on accessor methods would greatly improve the usability of JOPA.

Consider the example hierarchy in listing 4.1. Annotation `OWLDataProperty` is declared on the setter, and the field itself is left unannotated. The goal is to change the metamodel creation so there is no difference in created metamodels between the annotation directly on the field and the annotation on the method, as shown in the listing.

Therefore, the solution needs to be able to :

1. Correctly pair fields and their respective accessor methods
2. Use annotations from paired accessor methods while processing fields

```
@OWLClass(iri=Vocabulary.Foo)
public interface Foo {
    @OWLDataProperty(iri=Vocabulary.p_name)
    void setName(String name);
}

@OWLClass
public class Bar implements Foo {

    private String name;

    @Override
    public void setName(String name) {
        this.name = name;
        }
    }
```

Listing 4.1: Snippet showing annotation declared on setter and field to which the annotation would belong. Certain parts of code were omitted for brevity.

4.2.1 Pairing Fields and Accessors

Two distinct methods were considered for the first point- top-down and bottom-up approaches.

Top-down. During the field processing, if a field has no property mapping annotations, which would categorize the field into one of the categories from 2.7.2, it is saved in an array, where the value is a pair - class, and field.

After processing all entity classes and all their fields with this modification, a new phase of metamodel creation is added. In this phase, all entities are iterated over, along with their methods.

If a JOPA property annotation is found on the method, the method is checked if it is an accessor method. An exception is thrown if the annotated method is not an accessor method. If the method passes this check, a field name is extracted from the method name. Then, the class hierarchy is traversed top-down, using the hierarchy information stored in the metamodel, starting at the class the method belongs to.

During the traversal, all descendants are checked for fields with names that match the name extracted from the method. This checking is done through the array in which the unannotated fields were saved. If such a field is found, type compatibility[1] is checked. If the field is compatible, it is processed as usual, along with annotation from the method. The field is then removed from the array. If the field is not type-compatible with the method, an exception would be thrown.

After iterating over all entities, the array should be empty. If not, this means that there exist some unannotated fields. This is not allowed[2], and an

[1] Types are compatible if the field type is assignable from a return type, if the method is a getter, or argument if the accessor is a setter.

[2] All fields in entity classes must be annotated to be properly processed (see 2.7.2). Fields not mapped in an ontology must be explicitly marked as such (annotation *@Transient*, or one of *static, final* or *transient* field modifiers).

exception is to be thrown.

Bottom-up. During the processing of an entity class, a new phase is added before processing individual fields. This new phase traverses all methods of the entity class, searching for methods annotated with property mapping annotations. All found methods are checked to determine whether they are accessors, throwing an exception if otherwise. If the method passes the check, it is saved in an array, as a pair consisting of the entity class and the annotated method.

While processing fields, if the field has no annotation, a hierarchy is traversed bottom-up, where the root is the entity, the field belongs to.

All parents of the entity are searched for annotated methods through the array. If there is an accessor method, and its name corresponds to the name of the field, type compatibility is checked, and an exception is thrown if the types are not compatible.

If all checks are passed, the field is processed as usual, along with annotation from the method.

If no belonging method is found, an exception is thrown, for an unannotated field is not allowed in entity class in JOPA.

Detecting Multiple Accessors to One Field

Both methods also need to be able to detect when one field pairs with multiple annotated accessors. This is invalid, as that would mean that two or more distinct mappings exist to a single field (this rule is further discussed in section 4.2.2).

In the bottom-up approach, the solution is straightforward, as the approach is processing a single field at a time. A simple flag, if the accessor method was found, could be sufficient.

In the top-down approach, this is slightly more complicated as multiple fields are being processed at once, along with a single method. A solution could be a modification of the array, where each entry would have a simple flag to indicate whether the accessor method to the field has been found. Instead of deleting the item from the array, it would simply be marked.

Comparison

The comparison of the two methods yields quite a complete dualism.

- Top-down stores fields in the array, bottom-up methods
- Top-down traverses the hierarchy once for each method, bottom-up once for each field
- Top-down matches single method to multiple fields, bottom-up single field to multiple methods
- The time complexity of top-down grows linearly with the number of annotated methods, bottom-up with the number of unannotated fields

As the count of unannotated fields in a typical hierarchy should be higher than the count of annotated methods, the top-down method should be faster in most cases. However, time complexity is not a significant factor since a metamodel is built once at a startup. More important is the cohesion in the steps of the algorithm. The code that processes all fields at the same time/place is more readable and modifiable than the code that does it in two separate places.

Therefore, the bottom-up approach was chosen.

Extracting Field Name from Method

In order to correctly pair fields and their respective accessor methods, a way to extract the name of the field from the method (or vice versa) must be designed.

A strict set of rules for method/field naming must be used for extraction to be reliable and possible.

Fortunately, Java has extensively supported naming conventions, which can be used both for naming fields and accessors [38]. Only one addition should be applied - a getter for boolean value can be named `has<field name>`, as this form for getter is widely used and not in the convention. Table 4.1 contains examples of methods named accordingly to these conventions and field names extracted from them.

Package `java.beans` provides many tools centering around Java Beans[3]. Among them are tools enabling their introspection. Nevertheless, since there is no technical difference between Java Bean and Java class, `BeanInfo` can be used to introspect all Java classes [38].

With the use of `BeanInfo.getPropertyDescriptors` method, an array of information about properties[4] of a single class can be retrieved. This set of information includes the name of the property, along with accessor methods to this property and their names.

However, `BeanInfo` adheres strictly to the Java naming convention, and therefore does not recognize getters with `has` prefix.

With this disadvantage in mind, the decision was made to create a custom extractor to handle the `has` prefix too. This extraction consists of two steps, where the first step is removing the prefix, and the second step is decapitalizing the rest. Decapitalizing is not as trivial as setting the first letter to lowercase, for in special cases, when the second letter is also capitalized, the string is not modified. This is applied, for example, in the method `setURL`, as `setURL` is a setter for the field `URL`, not `uRL`. For this modified

[3] Java Beans are defined as "a reusable software component that can be manipulated visually in a builder tool"[38]. Java Beans are tightly coupled with the Java GUI framework AWT and are thought of as building blocks of applications. But at its core, Java Beans are simply Java classes with public fields and methods that can be reused in many different environments.

[4] In the context of Java Beans, properties are "named attributes associated with a bean that can be read or written by calling appropriate methods on the bean" - a class field with at least one public accessor method.

Method Name	Extracted Field Name
getFoo	foo
setURL	URL
isParked	parked
hasJoined	joined

Table 4.1: Table showing examples of accessor method names and field names extracted from them

decapitalizing, the method `Introspector.decapitalize` from the already mentioned `java.beans` package can be used.

4.2.2 Relaxing Model Constraints

In the current design, an exception would be thrown if multiple annotated methods were to belong to a single field.

Unfortunately, this would mean that creating any diamond-like hierarchy (hierarchy with a common predecessor) with an annotated method in a common predecessor[5] would throw an exception during metamodel creation. This is because, in the current bottom-up algorithm, one entity class will be visited multiple times if there exist more paths to this class in the hierarchy. Therefore, any annotated method in this class would be found and re-added to the field multiple times.

This would severely constrain the usability of the solution. Therefore, the algorithm should be modified so that finding the same annotated method to which the field belongs multiple times will not throw an exception.

How should this equality of annotated methods be defined? The annotated method consists of two parts - annotation and the method on which the annotation was declared. The need for equality of annotations is without discussion, but should the equality of methods be enforced too?

A simple hierarchy is shown in listing 4.2. This hierarchy is valid from the modeling perspective and represents a hierarchy that JOPA should support. However, the methods are not equal, so if equality was enforced on methods too, this hierarchy would not be valid in JOPA. Therefore equality of methods should not be enforced.

In conclusion, if more than one annotated method belonging to one field is found, the equality of annotations is checked. If all annotations are equal (their types and values are equal), then no exception should be thrown. If annotations differ, the hierarchy is considered invalid, and an exception should be thrown.

This relaxation of rules does not create any new potential ambiguity, that was not in the previous versions of the algorithm, as all annotations must agree.

[5] More precisely, if there exists an annotated method in ancestor to which exists two distinct paths in a hierarchy.

```
@OWLClass(iri = Vocabulary.Book)
interface Book {
    @OWLDataProperty(iri = Vocabulary.Title)
    void setTitle(String title);
}

@OWLClass(iri = Vocabulary.Recording)
interface Recording {
    @OWLDataProperty(iri = Vocabulary.Title)
    String getTitle();
}

@OWLClass(iri = Vocabulary.AudioBook)
class AudioBook implements Recording, Book {
    String title;
    /// code shortened
}
```

Listing 4.2: Pseudocode snippet showing hierarchy with two distinct methods belonging to one field. These methods are annotated by annotations that are equal.

However, this applies to ambiguity from annotations. What about ambiguity from methods? If equality is not enforced among them, could some ambiguity be created there?

Two parameters from methods are taken into account. A method name is used for pairing with fields. Any difference between a pair of methods in extracted names means that the methods belong to two distinct fields. Ambiguity is more potent with the method type[6]. Two different getters could have equal mappings and names, while returning different value types. However, this would raise an error during the compilation[7]. This error can be circumvented if the methods have different names, for example, two getters `isUsed` and `getUsed`, or one getter and one setter. This is, however, detected during metamodel creation. When a method with a name that corresponds to the field is found, its type compatibility is checked. If the types are not compatible (the method type is not a (sub)type of the field type), an exception is thrown.

Therefore, a degree of compatibility on both parameters used from methods is enforced, and any ambiguity is detected at the latest during metamodel creation.

[6] Type is the return type if the method is a getter or type of argument if the method is a setter.

[7] It is a compile-time error to declare two methods with the same name and parameters (override-equivalent signature) but different return types in one class. For more details, see [33].

Chapter 5

Implementation

After a solution was analyzed and designed, an implementation took place.

In this chapter, a few selected parts of the implementation are presented. These parts are just a small part of the whole implementation, and all changes can be seen in the pull requests to the JOPA GitHub repository[1].

5.1 Metamodel Hiearchy Changes

As discussed in 4.1.1, to accommodate the ability to reuse the current way of determining non-abstract descendants, the implementation needs to ensure that the entity type declared on an interface, is abstract from the JOPAs perspective. This is achieved by returning `true` on the method `AbstractIdentifiableType.isAbstract`. This method returns `true` if the type is abstract and `false` if otherwise.

Two approaches were proposed for ensuring that the value `true` is returned by the method `AbstractIdentifiableType.isAbstract` on types that are interface.

An easier way would be to use the current approach of JOPA. The current approach dynamically determines the abstractness of a class during the method call, using `Modifier.isAbstract` method.

The second method would be to change the class hierarchy to differentiate between the concrete (non-abstract) entity and interface (abstract) entity in the class hierarchy and override the method.

The second method was chosen because of the easier modifiability in the future.

The hierarchy changes can be seen in figure 5.1. The `EntityTypeImpl` was abstracted and split into two classes, one for concrete entity types, which are non-abstract, and the second one for abstract, non-instantiable entity types (such as interfaces or abstract classes). The method `isAbstract` in class `AbstractIdentifiableType` is now abstract and its implementation is left to child classes.

[1] `https://github.com/kbss-cvut/jopa/pull/160` and `https://github.com/kbss-cvut/jopa/pull/162` [visited on 23-05-17]

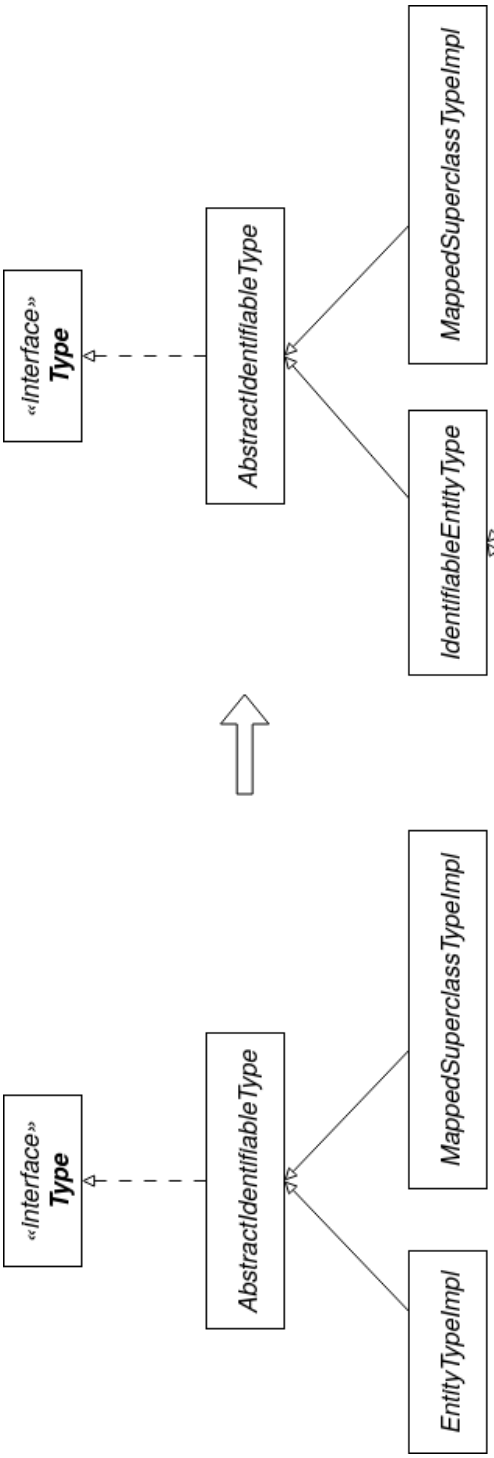


Figure 5.1: UML diagram of changes in the class hierarchy. Some classes and relations were omitted for brevity.

5.2 Using Annotations from Methods During Metamodel Creation

As designed in 4.2, annotations from methods must be used while processing paired fields.

However, it is imperative that the new implementation still supports fields with annotations declared directly on them. As processing a field and its annotations is a complex process, it would be impractical to create a new implementation for processing fields with annotations on methods that would exist alongside the current one.

A more suitable solution was chosen, which relies on a new interface `PropertyInfo`. It is a simple wrapper around a field or a field and accessor method. This interface defines a few methods which return data about a field, such as a name or a type, and a method that queries annotations of the field. This query is redirected to the method if annotations were declared on it or to the field if annotations were declared on the field. A shortened pseudocode of implementation can be seen in listing 5.1.

This interface is passed to all methods that process fields with annotations during metamodel creation. For these methods there is no difference between a field with annotations declared on a method or itself.

5.3 Comparing Annotations

As stated in 4.2.2, determining if two annotations are equal is crucial.

Although it is not possible to implement methods in the annotation interface[2], all classes in Java extend the `Object` class, and all annotation interfaces extend the interface `java.lang.annotation.Annotation`.

In this interface, among others, the method `equals` is defined. This method's behavior differs from the behavior of `Object.equals`, as it, along with type checking, recursively checks if all properties (members) are equal [39].

This is a straightforward and clean way to check equality on annotations, and as such, it is used in the implementation.

5.4 Documentation

Javadoc is a standard method of documenting code in Java and is widely used in JOPA.

Therefore, the main part of the documentation is written using Javadoc. Javadoc was primarily used to document the usage and behavior of new or modified classes and methods.

[2] The annotation interface is to annotation as the class is to the class instance. An annotation interface is a blueprint, a definition, whereas annotation is a concrete usage of an annotation interface.

```
public interface PropertyInfo {

    getAnnotation(annotationClass);

    getName();
}

class FieldInfo implements PropertyInfo {

    @Override
    getName() {return field.getName();}

    @Override
    getAnnotation(annotationClass) {
        return field.getAnnotation(annotationClass);
    }

}

class MethodInfo implements PropertyInfo {

    @Override
    String getName() { return field.getName();}

    @Override
    getAnnotation(annotationClass) {
        return method.getAnnotation(annotationClass);
    }
}
```

Listing 5.1: Pseudocode snippet demonstrating implementation of *PropertyInfo* wrapper

However, as Javadoc is primarily used for documenting code on individual elements (classes, fields, methods, etc.), and as such, it can be difficult to document complex features spanning several classes in it. Therefore, a candidate wiki page for JOPA's GitHub repository[3] was proposed. This documentation page discusses the usage of multiple inheritance in JOPA and contains a few code examples along with an explanation of the code. The proposed wiki page can be seen in appendix B.

[3] `https://github.com/kbss-cvut/jopa/wiki` [visited on 23-05-21]

Chapter 6

Evaluation

The evaluation of the solution consisted of two separate parts. At first, the existing suite of tests in JOPA was adjusted and extended to new behavior. Second, a demo application demonstrating functionality and usage of multiple inheritance in JOPA was created.

At the end of the chapter, the quality and execution of the implementation were evaluated.

6.1 JOPA Tests

The JOPA library contains an extensive suite of both unit and integration tests [40, 41]. Tests in JOPA use JUnit[1] and Mockito[2] frameworks, together with Hamcrest[3] library.

Some existing tests needed to be adjusted to adapt to the changes in the existing code and many tests were added to cover new parts of JOPA.

6.1.1 Changes in Existing Unit Tests

The implemented changes caused some of the existing tests to fail, some during compilation and others at runtime.

Most of the compilation errors were caused by passing a single AIT to the function `AbstractIdentifiableType.setSupertypes`, which, after changes accepts `Set<AbstractIdentifiableType>`. This was fixed by wrapping passed AIT in `Collections.singleton` method, which creates a set from a given element.

In the same category were compilation failures caused by the `PropertyInfo` wrapper around the fields (5.2).

In both cases, the solution was straightforward, although many tests were affected by these changes.

Some errors were caused by changes in logic. For example, the test `entityLoaderIgnoresInterfaceWithOwlClassAnnotation` originally veri-

[1] `https://junit.org`[visited on 23-05-05]
[2] `https://site.mockito.org/`[visited on 23-05-05]
[3] `https://hamcrest.org`[visited on 23-05-05]

```
@ParameterizedTest
@ValueSource(strings = {"isFoo", "getFoo", "hasFoo"})
void fromParsesBooleanIsGetterCorrectly(String methodName) throws
  NoSuchMethodException {
    Method genericGetter =
      MethodHolder.class.getDeclaredMethod(methodName);

    AnnotatedAccessor accessor =
      AnnotatedAccessor.from(genericGetter);


    assertEquals(genericGetter, accessor.getMethod());
    assertEquals("foo", accessor.getPropertyName());
    assertEquals(Boolean.class, accessor.getPropertyType());
}
```

Listing 6.1: Code snippet showing example of a parametrized test

fied that interfaces with annotation `OWLClass` were not considered to be manageable.

This test was reused to verify that interfaces with `OWLClass` annotation are now considered manageable.

6.1.2 New Unit Tests

Multiple unit tests were created to test the behavior of correctly processing fields while creating a metamodel. The focus was on the correct traversing of a metamodel hierarchy and the correct detection of ambiguities in it. As field processing is a process tightly integrated with multiple other classes, the use of class mocking was heavily leveraged [41].

Another thoroughly tested area was the `AnnotatedAccessor` class, where the extraction of a field name from a method is implemented. As this class is self-sufficient, no class mocks were needed. However, as there are many edge cases in name extraction, a great number of individual tests with a common flow but different inputs are needed. This is an ideal situation for parametrized testing, and it was used extensively. An example of such a test can be seen in listing 6.1.

6.1.3 Integration Tests

No changes were needed in integration tests, as the changes made to JOPA were backward compatible concerning the developer-facing part of JOPA.

The newly created integration test focused on a few areas:

- Correct saving of instances of classes with multiple parents into storage and consequent retrieving from storage.
- Correct polymorphism handling - one object can be found as an instance of all its parent classes.
- Correct handling of annotations on methods and their effect on fields.

During implementation, a form of Test Driven Development (TDD)[4] was used, as a first step in implementing a feature was to create an integration test that tested this new feature. This was useful, as JOPA is a library and therefore does not have any executable part (except the tests).

6.2 Demo Application

As part of the evaluation, two distinct class models that use multiple inheritance were provided.

The first hierarchy (figure 6.1) displays a hierarchy where the class `AudioBook` has two parents, `Book` and `Recording`. Both parents independently declare a single property. However, these two properties are equal in name and RDF mapping.

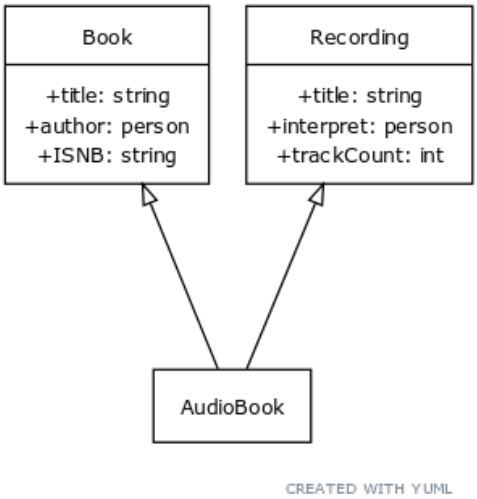


Figure 6.1: UML diagram of the class model with two parents containing two properties with equal names and mappings.

The second example contains a typical diamond hierarchy (figure 6.2). This means that a child class, `Copier`, inherits from two parents, `Scanner` and `Printer`, which share a common ancestor - class `Device`.

A simple application that enables CRUD operations over these two models was created[5]. The application provides REST API for data operations, which is used by a set of Postman[6] tests. These tests were created to test and showcase the application.

[4] TDD is a software development process where tests for new functionality are created before the new functionality. These tests should fail, as the functionality is not yet implemented. After this, a feature is implemented as fast as possible and the tests should pass successfully. When all tests are passing, code refactoring follows before the process repeats [41].

[5] Available from `https://github.com/Kulda22/jopa-multiple-inheritance-demo` [visited on 23-05-17]

[6] `https://www.postman.com/` [visited on 23-05-06]

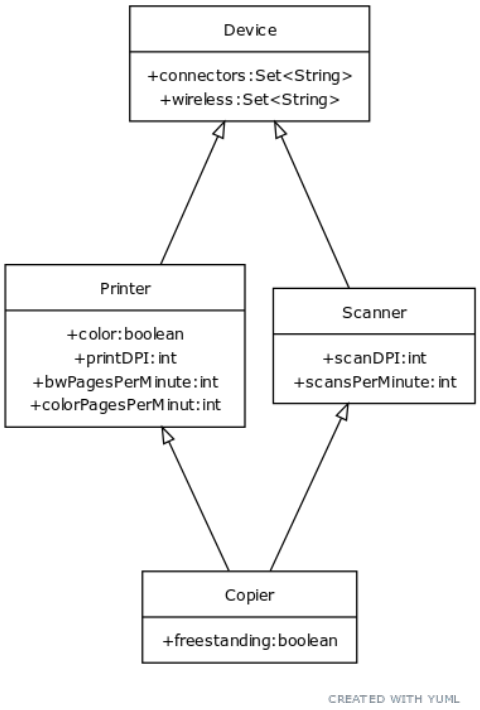


Figure 6.2: UML diagram of the class model with diamond hierarchy.

This demo application demonstrates the usage and capabilities of multiple inheritance in JOPA. One of the prominent features of (multiple) inheritance is polymorphism, whose effect can be demonstrated in multiple ways. The first one is that the same `AudioBook` can be found by querying the application for a `Book` by its ISBN or by querying for `Recording` by its URI. The second one is that, if, for example, the endpoint for reading all objects of the class `Device` is queried, all instances of classes `Device`, `Printer`, `Scanner`, and `Copier` are returned. This is because all copiers are also devices, and so on.

Emulating this behavior with two or more parents without multiple inheritance would be difficult. Emulating with single inheritance only would require a significant amount of unnecessary code and manual SPARQL queries, and any change in the hierarchy would demand extensive rewrites in the source code.

However, the demo shows a considerable weakness in the chosen solution. In both examples, all parent entity classes are interfaces, as it is required[7] for the hierarchies to be created in JOPA. Therefore, any parent cannot be instantiated directly, as they are only interfaces. That means that instances of class `Book` cannot be created, only `AudioBook`.

This can be mitigated by creating a special entity class for each interface parent, which shares the same class IRI and implements only the parent interface. If there is a need for an instance of the interface parent, an instance

[7] Only one of *Scanner* or *Printer* entity classes in the diamond hierarchy could be a class, not an interface.

of this special class can be used instead. This is a simple and effective solution. On the other side, it forces the creation of new boilerplate classes. Therefore, it should not be used automatically, only when needed.

6.3 Code Maintainability

After implementing tests and creating the demo application, the decision was made to support more annotations on methods. So far, only property mapping annotations (`OWLObjectProperty`, `OWLAnnotationProperty`, and `OWLDataProperty`) were considered. However, the evaluation so far demonstrated that this list should be extended with `ParticipationConstraints`, `Convert`, `Enumerated` and `Sequence` annotations, as they are exclusively used with property mapping annotations.

Excluding them would lead to needless code repetition and potential ambiguities.

One of the metrics of code quality is maintainability, and one part of the maintainability of the code is the amount and complexity of changes needed to implement a new feature [42, 43]. As enabling a new set of annotations to be declared on methods can be looked at as a new feature, it can be used to evaluate the modifiability of implementation.

Only two parts of the code had to be modified to implement the new feature.

To specify where can the given annotation be declared, annotation can be annotated by annotation `@Target`, which specifies in which context the given annotation is applicable [44]. This had to be modified to include the usage of annotation on methods.

The second modification was adding the newly supported annotations to the method, which checks if all annotations in two different methods are equal during hierarchy traversal to prevent ambiguities.

Along with these two code changes, new tests were implemented to test the new behavior.

As these changes were minimal in scope and complexity, it can be declared that the code is maintainable.

Chapter 7

Conclusion

As stated in chapter 1, this thesis aimed to:

- Implement multiple inheritance in JOPA
- Prevent errors caused by differences between JOPA and OWL domain models
- Be fully backward compatible

As demonstrated by the tests in JOPA and the demo application, in chapter 6, the implementation successfully enabled the usage of multiple inheritance in JOPA.

Runtime errors caused by unchecked access to properties of parents of the class can now be prevented. With JOPA supporting multiple inheritance, a plethora of errors caused by the differences between domain models can now be prevented, as developers can now leverage Java's powerful type system while working with properties declared in class parents.

The critical point is that all changes are backward compatible. No change in the source code of applications using JOPA will be required due to the new features. This is because from the developer's point of view only two things changed:

- Interfaces can now be used as entity classes
- Annotations can be declared on methods

In conclusion, it can be stated that all the goals of the thesis were achieved successfully.

7.1 Future work

Even though multiple inheritance was implemented successfully and completely, more work could be done to enhance the capabilities of the whole JOPA library using work done in this thesis.

One is the support for more JOPA annotations on accessor methods. Annotations such as `@Transient`, or `@Sparql`, could benefit from this option, if the need for it ever arises.

Another possible feature would be the option to define whether the field during (de)serialization should be accessed directly, or by its accessor method, as supported by JPA [16]. This feature could re-use a substantial part of the work done in this thesis.

Implementation of above-mentioned features, or any others, depends solely upon the needs of developers using JOPA in their applications.

Bibliography

1. BERNERS-LEE, Tim; HENDLER, James; LASSILA, Ora. The Semantic Web. *Scientific American Magazine*. 2001, vol. 2001, no. 284, pp. 28–37.
2. *Semantic Web* [online] [visited on 2022-12-14]. Available from: `https://www.w3.org/standards/semanticweb/`.
3. MOTIK, Boris; PATEL-SCHNEIDER, Peter; PARSIA, Bijan. *OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax (Second Edition)* [online] [visited on 2022-12-30]. Available from: `https://www.w3.org/TR/owl2-syntax/`.
4. HITZLER, Pascal; KRÖTZSCH, Markus; PARSIA, Bijan; PATEL-SCHNEIDER, Peter F.; RUDOLPH, Sebastian. *OWL 2 Web Ontology Language Primer (Second Edition)* [online] [visited on 2022-12-10]. Available from: `https://www.w3.org/TR/owl2-primer/`.
5. SCHREIBER, Guus; RAIMOND, Yves. *RDF 1.1 Primer* [online] [visited on 2022-12-10]. Available from: `https://www.w3.org/TR/rdf11-primer/`.
6. ANTONIOU, Grigoris; HARMELEN, Frank van. *A semantic web primer*. Third Edition. Cambridge: MIT Press, c2012. ISBN 978-026-2012-423.
7. KOTRLÝ, Zdeněk. *Mapování atributů založené na SPARQL dotazech v knihovně JOPA*. Prague, 2021. Available also from: `https://dspace.cvut.cz/handle/10467/94516`. Bachelor Thesis. Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Computer Science.
8. BRICKLEY, Dan; GUHA, R.V. *RDF Schema 1.1* [online] [visited on 2022-12-10]. Available from: `https://www.w3.org/TR/rdf-schema/`.
9. LEDVINKA, Martin. *Leveraging Semantic Web Technologies in Domain-specific Information Systems*. Prague, 2020. Available also from: `https://dspace.cvut.cz/handle/10467/91091`. Dissertation Thesis. Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Computer Science.
10. SIMONS, Peter. *Ontology* [online] [visited on 2022-12-18]. Available from: `https://www.britannica.com/topic/ontology-metaphysics`.

11. GRUBER, Thomas R. Toward principles for the design of ontologies used for knowledge sharing? *International Journal of Human-Computer Studies*. 1995, vol. 43, no. 5-6, pp. 907–928. ISSN 10715819. Available from DOI: 10.1006/ijhc.1995.1081.

12. GUARINO, Nicola; OBERLE, Daniel; STAAB, Steffen. What Is an Ontology? *Handbook on Ontologies*. 2009, pp. 1–17. ISBN 978-3-540-70999-2. Available from DOI: 10.1007/978-3-540-92673-3_0.

13. PRUD'HOMMEAUX, Eric; SEABORNE, Andy. *SPARQL Query Language for RDF* [online] [visited on 2022-12-16]. Available from: https://www.w3.org/TR/rdf-sparql-query/.

14. BECKETT, Dave. *What does SPARQL stand for?* [online] [visited on 2022-12-16]. Available from: https://lists.w3.org/Archives/Public/semantic-web/2011Oct/0041.html.

15. ELLIOTT, James; FOWLER, Ryan; O'BRIEN, Tim. *Harnessing Hibernate*. Sebastopol: O'Reilly, 2008. ISBN 05-965-1772-6.

16. JCP. *JSR 317: Java^{TM} Persistence API, Version 2.0*. 2009. Technical report. Java Community Process.

17. KŘEMEN, Petr; KOUBA, Zdeněk. Ontology-Driven Information System Design. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*. 2012, vol. 42, no. 3, pp. 334–344. ISSN 1094-6977. Available from DOI: 10.1109/TSMCC.2011.2163934.

18. LEDVINKA, Martin; KŘEMEN, Petr. JOPA: Stay Object-Oriented When Persisting Ontologies. In: *Enterprise Information Systems*. Cham: Springer International Publishing, 2015, pp. 408–428. ISBN 978-3-319-29132-1. Available from DOI: 10.1007/978-3-319-29133-8_20.

19. FOWLER, Martin. *Patterns of enterprise application architecture*. Boston: Addison-Wesley, c2003. ISBN 03-211-2742-0.

20. WEGNER, Peter. Panel on inheritance: Varieties of Inheritance. *ACM SIGPLAN Notices*. 1988, vol. 23, no. 5, pp. 35–40. ISSN 0362-1340. Available from DOI: 10.1145/62139.62142.

21. SCHÄRLI, Nathanael; DUCASSE, Stéphane; NIERSTRASZ, Oscar; BLACK, Andrew P. Traits: Composable Units of Behaviour. *ECOOP 2003 – Object-Oriented Programming*. 2003, pp. 248–274. ISBN 978-3-540-40531-3. Available from DOI: 10.1007/978-3-540-45070-2_12.

22. TRUYEN, Eddy; JOOSEN, Wouter; VERBAETEN, Pierre; JØRGENSEN, Bo Nørregaard. A Generalization and Solution to the Common Ancestor Dilemma Problem in Delegation-Based Object Systems. 2004.

23. STROUSTRUP, Bjarne. Multiple Inheritance for C++. *C/C++ users journal*. Vol. 1999. ISSN 1075-2838.

24. BARRETT, Kim; CASSELS, Bob; HAAHR, Paul; MOON, David A.; PLAYFORD, Keith; WITHINGTON, P. Tucker. A monotonic superclass linearization for Dylan. *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 1996, pp. 69–82. ISBN 089791788X. Available from DOI: 10.1145/236337.236343.

25. Multiple Inheritance in C++. *Programming with Objects*. 2003. ISBN 9780470547144. Available from DOI: 10.1109/9780470547144.ch16.

26. GOSLING, James; HSU, Hansen; WEBER, Marc. *Gosling, James oral history, part 1 of 2*. Mountain View, CA: Computer History Museum, 2019.

27. GOSLING, James. Java: an Overview: the original Java whitepaper. 1995.

28. MARTIN, Robert C. Java and C++: a critical comparison. *Java Gems: jewels from Java Report*. 1998th ed., pp. 51–68.

29. More OO in Java—Interfaces and Abstract Classes. *Foundations of Java for ABAP Programmers*. 2006, pp. 57–60. ISBN 978-1-59059-625-8. Available from DOI: 10.1007/978-1-4302-0140-3_12.

30. CHARATAN, Quentin; KANS, Aaron. Java: Interfaces and Lambda Expressions. In: *Programming in Two Semesters*. Cham: Springer International Publishing, 2022, pp. 455–479. ISBN 978-3-031-01325-6. Available from DOI: 10.1007/978-3-031-01326-3_19.

31. DATHAN, Brahma; RAMNATH, Sarnath. Exploring Inheritance. In: *Object-Oriented Analysis, Design and Implementation*. Cham: Springer International Publishing, 2015, pp. 223–273. ISBN 978-3-319-24278-1. Available from DOI: 10.1007/978-3-319-24280-4_9.

32. GRAND, Mark. *Java: Language Reference*. Second Edition. O'Reilly Media, 1997. ISBN 978-1565923263.

33. GOSLING, James; JOY, Bill; STEELE, Guy; BRACHA, Gilad; BUCKLEY, Alex. *The Java Language Specification: Java SE 8 Edition*. 5th ed. Upper Saddle River, NJ: Addison-Wesley, 2014. ISBN 978-0-13-390069-9.

34. GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Design patterns: elements of reusable object-oriented software*. Boston: Addison-Wesley, 1995. ISBN 978-0201633610.

35. ANCONA, Davide; LAGORIO, Giovanni; ZUCCA, Elena. Jam - designing a Java extension with mixins. *ACM Transactions on Programming Languages and Systems*. 2003, vol. 25, no. 5, pp. 641–712. ISSN 0164-0925. Available from DOI: 10.1145/937563.937567.

36. LOCKE, Jonathan. *The KivaKit Manifesto — A New Vision for Java* [online] [visited on 2022-12-11]. Available from: https://medium.com/the-techlife/kivakit-a-new-vision-for-java-66e7f6b18ae6.

37. COLYER, Adrian. *Eclipse AspectJ: aspect-oriented programming with Aspectj and the Eclipse Aspectj development tools*. Upper Saddle River: Addison-Wesley, c2005. ISBN 03-212-4587-3.

38. HAMILTON, Graham. *JavaBeans^{TM}: API specification*. Version 1.01-A. Sun Microsystems, 1997.

39. *Java^{TM} Platform, Standard Edition 8 API Specification: Interface Annotation* [online] [visited on 2023-05-05]. Available from: `https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Annotation.html`.

40. CRISPIN, Lisa; GREGORY, Janet. *Agile testing: a practical guide for testers and agile teams*. New Jersey: Addison-Wesley, c2009. ISBN 978-0-321-53446-0.

41. KENT, Beck. *Test Driven Development: By Example*. 1st. Addison-Wesley Professional, 2002. ISBN 978-0321146533.

42. COLEMAN, Ron. Beauty and Maintainability of Code. *2018 International Conference on Computational Science and Computational Intelligence (CSCI)*. 2018, pp. 825–828. ISBN 978-1-7281-1360-9. Available from DOI: `10.1109/CSCI46756.2018.00165`.

43. VISSER, Joost; RIGAL, Sylvan; LEEK, Rob van der; ECK, Pascal van; WIJNHOLDS, Gijs. *Building Maintainable Software, Java Edition: Ten Guidelines for Future-Proof Code*. 1st edition. O'Reilly Media, 2016. ISBN 9781491953495.

44. *Java^{TM} Platform, Standard Edition 8 API Specification: Annotation Type Target* [online] [visited on 2023-05-05]. Available from: `https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Target.html`.

Appendix A

Supported annotations on accessors

Table A.1 denotes which annotations can be used on accessor methods.

JOPA Annotation	Supported on accessor methods
@Asserted	No
@Convert	Yes
@Enumerated	Yes
@Id	No
@Individual	No
@Inferred	No
@OWLAnnotationProperty	Yes
@OWLDataProperty	Yes
@OWLObjectProperty	Yes
@ParticipationConstraints	Yes
@Properties	No
@Sequence	Yes
@Sparql	No
@Transient	No
@Types	No

Table A.1: Table denoting which field annotations can also be declared on accessor methods

Appendix B

Proposed New Documentation Page

Within the scope of the thesis, a new documentation page concerning multiple inheritance in JOPA was created. This page will be added to the wiki section of the GitHub repository of JOPA[1].

B.1 JOPA and Multiple Inheritance

Multiple inheritance is now supported in JOPA. Please see the text below for details and usage.

B.1.1 Usage

As Java does not directly support multiple inheritance, an entity class cannot extend multiple entity classes (or mapped superclasses) at once.

JOPA takes advantage of interfaces, as a class can implement multiple interfaces simultaneously. In order to support multiple inheritance, it is possible to declare interfaces as entity classes in JOPA with `@OWLClass` annotation, the same as plain Java classes. A child class can then implement multiple of these interface entity classes, along with almost all features of single inheritance.

The use of interface entity classes has some caveats that emerge from Java interface constraints.

- It cannot be directly instantiated, as it is an interface.
- It cannot declare any non-static non-final properties. Child classes cannot inherit any attributes from these entity classes.

The second caveat is partly omitted by using property annotations on methods.

Property Annotations on Methods

In order to reduce code duplication and mistakes from duplication, JOPA supports declaring field annotations (`OWLDataProperty`, `OWLObjectProperty`,

[1] Please note that some changes have been made, as the original was written in a Markdown markup language.

```
@OWLClass(iri = Vocabulary.CLASS_BASE + "AParentI")
interface AParentI {}

@OWLClass(iri = Vocabulary.CLASS_BASE + "BParentI")
interface BParentI {}

@OWLClass(iri = Vocabulary.CLASS_BASE + "InterfaceChild")
class InterfaceChild implements AParentI, BParentI {
    @Id
    private URI uri;
}
```

Listing B.1: Simple hierarchy with multiple inheritance

`OWLAnnotationProperty`, `ParticipationConstraints`, `Convert`, `Enumerated` and `Sequence`) on getters and setters.

This means that the property annotation, along with IRI and other parameters, can be declared once in the parent entity class. All non-abstract children entity classes then only need to implement the method and declare the field without any annotations, as the field will *inherit* property annotation from the annotated method.

The assignment of the field to annotated method starts when an unannotated field is discovered during metamodel creation. The hierarchy is searched bottom-up, looking for annotated methods in parent entity classes. If an annotated method is found, the field name is extracted from the method name based on Java naming conventions. If the name matches the name of the field, the field is processed with property annotations from the method.

B.1.2 Examples

Simple Hierarchy

In code example B.1, a simple hierarchy is defined. This hierarchy mirrors some arbitrary ontology. In Java's single inheritance, a decision would have to be made, which class would be a parent, and which would be omitted from the hierarchy. However, this would disconnect the Java model from the model in underlying storage, which is problematic.

This example shows the power of polymorphic behavior. Storage can be queried using the `<T> T EntityManager.find` method. EntityClass argument is a Java class to look for and return. With multiple inheritance, we can pass any of the three classes `InterfaceChild`,`BParentI` and `AParentI`, with valid identifier and have returned instance of `InterfaceChild`, as in example B.2.

Annotated Methods

The hierarchy from listing B.3 contains field `foo` which *inherits* property annotation from method `getFoo()`.

```
final InterfaceChild child = new InterfaceChild();
child.setId(id);

em.persist(child);

OWLChildClassA foundChild = em.find(InterfaceChild.class, id);

AParentI parentBFound = em.find(AParentI.class, id);

BParentI parentAFound = em.find(BParentI.class, id);
```

Listing B.2: Snippet demonstrating polymorphic behavior

```
@OWLClass(iri = Vocabulary.BarInterface)
public interface BarInterface {
    @OWLDataProperty(iri = Vocabulary.foo)
    String getFoo();
}

@OWLClass(iri = Vocabulary.BarChild)
public class BarChild implements BarInterface {
    protected String foo;

    @Override
    public String getFoo() {return foo;}
}
```

Listing B.3: Code snippet demonstrating annotations on methods

Two Parents With Same Properties

A field can *inherit* annotations from multiple methods as long as the annotations are equal (shown in listing B.4). If annotations are not equal, an exception is thrown during metamodel creation.

Other Examples

For more examples, see integration tests in `MultipleInheritanceTestRunner.java` in `/jopa-integration-tests/src/main/java/cz/cvut/kbss/jopa/test/runner` or JOPA examples (`https://github.com/kbss-cvut/jopa-examples`) repository.

B.1.3 Notes

Mixing field annotations and method annotations

Do not mix field and method annotations on a single property. Field annotations SHOULD overwrite the method annotations, but they are considered undefined behavior.

```
@OWLClass(iri = Vocabulary.ParentA)
public interface ParentA {
    @OWLDataProperty(iri = Vocabulary.name)
    void setName(String name);
}

@OWLClass(iri = Vocabulary.ParentB)
public interface ParentB {
    @OWLDataProperty(iri = Vocabulary.name)
    void setName(String name);
}

@OWLClass(iri = Vocabulary.BarChild)
public class Child implements ParentA,ParentB {
    protected String name;

    @Override
    public void setName(String name){this.name = name;}
}
```

Listing B.4: Code snippet demonstrating multiple methods with annotations belonging to one field