UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Compile time semantic analysis and verification of conceptual models in application to transactional business information systems

Author:
Vladyslav BILYK

Supervisor:
Oles HODYCH, PhD

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Department of Computer Sciences
Faculty of Applied Sciences

Lviv 2022

Declaration of Authorship

I, Vladyslav BILYK, declare that this thesis titled, "Compile time semantic analysis and verification of conceptual models in application to transactional business information systems" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

Compile time semantic analysis and verification of conceptual models in application to transactional business information systems

by Vladyslav BILYK

Abstract

Conceptual modeling of information systems is an approach to systems analysis and design that is widely-used in the field of software engineering. Its aim is to obtain a description of the general knowledge that an information system needs to know, which is called a conceptual schema. Although, the information about a conceptual schema is widely used by various technologies in a programmatic way for automation of error-prone software engineering tasks, there is a lack of refined metamodeling facilities that provide domain discoverability at design time. This forces software engineers to turn to awkward ways of representing metadata, which results in unreliable systems with limited evolvability. In this work, we develop a technology for compile time semantic analysis that captures the description of a conceptual schema in a form of metadata modeled in the source code. We focus on the Java programming language in particular, taking advantage of its annotation processing capabilities. We provide our implementation in the context of a particular framework – Trident Genesis. As a means of evaluation we employ an approach of qualitative research by conducting an experiment targeted at a focus group of software engineers. We demonstrate that our findings indicate a definitive improvement in domain discoverability, system reliabilty and evolvability. The core principles of the presented implementation lay the groundwork for the development of a general metamodeling abstraction.

Acknowledgements

I would like to thank the following for their help in connection with this thesis:

My thesis supervisor, Oles Hodych, for his valuable feedback and support throughout this project.

The software engineering team at Trident Genesis for participating in the experiments.

Armed Forces of Ukraine for protecting our country and our freedom.

Contents

Declaration of Authorship **i**

Abstract **ii**

Acknowledgements **iii**

1 Introduction **1**
1.1 Domain modeling 2
1.2 System reliability 2
1.3 System evolvability 3
1.4 Technical approach 5
1.5 Qualitative research 6
1.6 Structure 6

2 Background **7**
2.1 Conceptual modeling of software systems 7
2.2 Domain-driven design and its terminology 8
2.3 Java programming language 9
2.4 Java reflection 9
2.5 Java annotations 9
2.6 Annotation processing 10

3 Related work **12**
3.1 C# nameof 12
3.2 Java nameof hack 13
3.2.1 Method references 14
3.2.2 Reflection and byte-code manipulation 14
3.2.3 The hack 14
3.3 Project Lombok 14
3.3.1 @FieldNameConstants annotation 15
3.4 Hibernate Metamodel Generator 16

4 Implementation details **18**
4.1 Entity graph 19
4.2 Meta-model generation algorithm 19
4.3 Meta-model structure 22
4.4 Usage example 25

5 Evaluation **26**
5.1 Discoverability of the domain model 27
5.2 Reliability 28
5.3 Evolvability 29
5.4 Performance 29
5.5 Correctness of the generation mechanism 30

5.6 Intuitiveness and ease of use 31

6 Conclusion and future work **33**

6.1 Review against the original objectives 33

6.2 Future work 33

6.2.1 Integration with IDEs 33

6.2.2 Compile time constant values 34

6.2.3 Support for external metadata 34

6.2.4 General framework indepent approach 34

Bibliography **35**

List of Figures

2.1 UML class diagram for a simplified banking domain 7
2.2 UML class diagram illustrating an additive change to the conceptual model 8
2.3 UML class diagram illustrating a breaking change to the conceptual model 8
2.4 Compilation process of javac 10

3.1 Lombok Annotation Processor modifies an AST 15

4.1 Input and output of the domain model processor 18
4.2 `Person` entity graph 19
4.3 A high-level view of the meta-model generation algorithm 20
4.4 The procedure of generating a meta-model 21
4.5 An entity meta-model graph where `Person` and `User` are metamodeled, but `Vehicle` is not 22
4.6 A meta-model graph for entity `Person` after `Vehicle` was metamodeled 22
4.7 UML class diagram for generated meta-models 23
4.8 Traversing the entity graph with the help of Eclipse IDE code auto-completion feature 25

5.1 Question 1: Domain discoverability 27
5.2 Question 2: Domain discoverability 27
5.3 Question 3: Domain discoverability 28
5.4 Question 4: Reliability 28
5.5 Question 5: Evolvability 29
5.6 Question 6: Performance 29
5.7 Question 7: Correctness 30
5.8 Question 8: Correctness 30
5.9 Question 9: Correctness 31
5.10 Question 10: Intuitiveness and ease of use 31
5.11 Question 11: Intuitiveness and ease of use 32
5.12 Question 12: Intuitiveness and ease of use 32

List of Abbreviations

APT	**A**nnotation **P**rocessing **T**ool
AST	**A**bstract **S**yntax **T**ree
ORM	**O**bject-**R**elational **M**apping
OOP	**O**bject-**O**riented **P**rogramming

Chapter 1

Introduction

An important piece of work in the information systems field by Antoni Olive [Oli07] defines the term *conceptual modeling* as the activity of describing and structuring the general knowledge a particular information system needs to know. The main objective of conceptual modeling is to obtain that description, which is called a *conceptual schema*. Conceptual modeling is an important part of requirements engineering, the first and most important phase in the development of an information system. However, any information system will invariably undergo modification, either due to the demands of its users or as a result of a change in the nature of the information itself, or for other reasons. Therefore, one of the most important properties of information systems is evolvability.

An ability to access and process the information about a conceptual schema programmatically at runtime is widely used by various technologies for automation of otherwise complex and error-prone software engineering tasks. For example, Object-Relational Mapping (ORM)[1] technologies automate mapping between two representations of a conceptual schema: expressed in OOP terms as classes and expressed in terms of a relational database. This information is a structured representation of a conceptual schema, which we shall refer to as *meta-model*. Although accessible at runtime, such meta-models also need to be made available to software engineers at design time. The information provided by meta-model is called *metadata*. Since this meta-level information is also a part of a system it is desirable for it to be evolvable as well. Moreover, due to the fact that a meta-model reflects the structure of a conceptual schema, it must be consistent with it. Ideally, a meta-model would always be automatically updated to match the structure of a conceptual schema.

The aim of this research is to develop a technology for compile time semantic analysis that would capture the description of conceptual models in a form of metadata directly represented in the source code. We outline the following objectives:

1. Improve *domain modeling* efficiency and provide advanced *domain discoverability* features at design time.

2. Improve system reliability and ensure correctness of references to domain models at compile time.

3. Improve system evolvability.

The key result of this research is the developed technology itself – an approach to achieve the stated objectives.

[1]Object-Relational Mapping is a technique used to interact with a database through an interface implemented in object-oriented paradigm.

1.1 Domain modeling

Any software system aims to address a specific problem. The area surrounding this problem is known as the problem domain. *Domain modeling* is a form of conceptual modeling commonly used in software engineering. Its aim is to construct a *domain model* – a structured representation of the problem domain with a description of core concepts and their relationships. We define *domain discoverability* as the ability to *discover* the domain model, that is, to inspect the conceptual schema of the domain. We later show that domain discoverability can be limited by the programming language used to write the software.

Efficient software construction requires a certain kind of knowledge to be present in the mind of a software engineer. Peter Naur [Nau85] conveyed an insightful idea of Theory Building View of programming, stating that a program is a shared mental construct that lives in the minds of the people who work on it. Therefore, the programmer possessing the theory is able to respond constructively to any demand for a modification of the program. The same can be said about domain modeling where the general knowledge of the domain needs to be present at all times in order to develop correct models. However, that might prove to be a difficult task for a domain of significant size, requiring the software engineers to frequently brush up their knowledge of the target domain through the use of a secondary source of information, such as documentation or source code of the software. In order to tackle this difficulty many supporting tools have been developed. For example, modern IDEs[2] come with a handful of advanced features, such as code auto-completion, which is a practical way of enhancing domain discoverability.

Brooks [FPB86] defines two kinds of complexity involved in the process of software construction: essential and accidental. Essential complexity stems from the inherent properties of software, such as the size of software systems, conformity to other interfaces, changeability and invisibility (inability to be visualized). Accidental complexity is manifested in any activity that engages in representation of conceptual software structures in programming languages and mapping of those structures onto machine languages within space and speed constraints. Among past breakthroughs in solving accidental difficulties are: high-level programming languages, time-sharing systems and unified programming environments. Following Brooks's excerpt on time-sharing which he views as an attack on a specific accidental difficulty – interruption of consciousness due to a need to call for compilation and execution, which might result in the decay of grasp of all that is going on in a complex system – our main hypothesis is that the lack of programming language-level metamodeling facilities with advanced design time domain discoverability features is the same kind of difficulty with negative effects of similar nature.

1.2 System reliability

Reliability of a system is its ability to perform a given task in an expected way without causing errors. In order to build reliable systems it is important to understand those systems. Moseley & Marks [BM06] identified two widely-used approches to

[2]Integrated Development Environment (IDE) is software that combines common developer tools into a single graphical user interface.

understanding systems: *testing* and *informal reasoning*; with an emphasis on the importance of the latter. Informal reasoning is an attempt to understand the system by examining it from the inside. Its importance was explained by Moseley & Marks by the fact that improvements in informal reasoning lead to *less errors being created*, as opposed to improvements in testing that lead only to *more errors being detected*. One of the desirable properties of testing is high code coverage, which is a measure of the amount of source code of a program that was executed, hence covered, during testing. However, for systems of significant size, characterized by large code volume with complex structure, testing is usually a time-consuming process. No guarantees can be given that a failing test will not occur at the very end of the whole process. Because of this, understanding systems through testing is also more time-consuming. Therefore, it is preferable to focus on ways of improving informal reasoning.

When we examine a system from the inside we do it at design time, that is, during the phase of system construction. For example, reasoning about the source code of a system's component is an activity carried out at design time. Modern IDEs blur the line between design time and compile time, seemlessly integrating the compilation process into design time in order to increase developer productivity. This allows software engineers to benefit from messages signaled by a compiler, making a great contribution to improvements in informal reasoning. Therefore, delegating as much system validation as possible to a compiler will result into more reliable systems.

1.3 System evolvability

All systems are prone to change, especially the successful ones. This is due to the fact that software that is found to be useful is often pushed to its limits by its users who invent new uses for it. Change management thus plays an essential role in the lifecycle of a software system. Whenever a change is introduced to the domain model, all parts of a system that interact with the changed component must be verified and modified if necessary. While the verification is usually covered by automated tests, the modification of related components has to be carried out by a software engineer. In order to ensure that the latest change is adopted correctly a software engineer must know the exact locations of those components in the source code. Once again, this can be a difficult task when working with a large system and that is why the role of a compiler is critical, for it can inform the software engineer of those places in the source code.

The underlying assumption is that software is being developed in a compile time safe manner. However, given the general purpose nature of modern programming languages, this assumption is not always true. There is a lack of a language-level abstraction that could be used as domain model metadata, that carries type information, to reference the conceptual model. This limitation forces software engineers to use textual string representations instead, that are "hard-coded" into the program. This is known to be unreliable because it is not the responsibility of a compiler to validate the contents of a string. A program deemed valid by a compiler might make use of incorrect metadata that results in a runtime error. Although, were the metadata represented in the form of a meta-model instead, all rules of compile time validation would be applicable to it. Moreover, any modification to the domain model would be reflected in the meta-model. Therefore, it would be possible to efficiently track the related components in need of modification at design time due to messages signaled by a compiler.

To illustrate a possible case where a meta-model might be required we provide the following example. Consider a simplified domain that consits of a single concept called `Customer`:

```
class Customer {
    private String name;
    private int age;
}
```

LISTING 1: A java class for the `Customer` concept.

The following listing illustrates the construction of a database query to fetch all customers who are over 21:

```
String query = "SELECT name FROM customers WHERE age >= 21;";
```

LISTING 2: SQL query with hard-coded metadata that fetches the names of all customers of age over 21.

The problem with this code is that it uses hard-coded metadata. A compiler can't tell whether `name` and `age` are parts of the `Customer` concept. It also has no way of verifying whether `customers` is a valid database table. As a result, this code is unreliable and difficult to maintain. It is easy to imagine that some time in the future the conceptual schema might change, leading to the `Customer` concept no longer having the attribute `name`, but `fullName` instead. Consequently, each such occurence of hard-coded metadata must be manually located throughout the whole system.

It is true that using raw strings to construct SQL queries is a bad practice and better approaches have been developed, such as ORM frameworks. However, the core issue still remains, as demonstrated by the following example:

```
QueryModel<Customer> query = select(Customer.class).where()
    .prop("age").gt().val(21)
    .yield().prop("name")
    .model();
```

LISTING 3: SQL query from Listing 2 expressed using an ORM framework.

The `"age"` and `"name"` strings still must be used to refer to `Customer` attributes.

Now, consider an approach utilizing a meta-model:

```
QueryModel<Customer> query = select(Customer.class).where()
    .prop(Customer_.age).gt().val(21)
    .yield().prop(Customer_.name)
    .model();
```

LISTING 4: SQL query from Listing 3 using a meta-model.

Here `Customer_` is a meta-model class. It guarantees that the database query is constructed in a compile time safe manner, since a change to the domain model is immediately reflected in the meta-model. It also makes the system more evolvable. In case a breaking change took place, an appropriate compilation error would follow.

Ideally, a meta-model would also capture the relationships between concepts, allowing the software engineer to traverse the domain graph in the source code. Expanding the previous example, consider a new concept called `Order` and its relationship to `Customer`:

```
class Order {
    private int number;
}

class Customer {
    private String name;
    private int age;
    private Order order;
}
```

LISTING 5: A java class for the `Order` concept.

Then, the relationship between `Order` and `Customer` could be captured by a meta-model:

```
String path = Customer_.order.number; // "order.number"
```

LISTING 6: A meta-model capturing the relationship between `Customer` and `Order`.

This would make a powerful addition to the domain modeling capabilities of a programming language.

1.4 Technical approach

Taking into account the widespread adoption and use of object-oriented programming in domain-driven design, we focus on the Java programming language in particular. Since Java language specification does not support class meta-models, we provide our own implementation of a meta-model generation mechanism. The implementation is based on a feature of Java – annotations, supplemented by annotation processing – an ability to process annotations at compile time.

The implementation we provide is designed with a particular software development technology in mind – Trident Genesis[3] (TG). The choice was made to integrate the implementation with the surrounding framework in order to make the development process manageable in terms of time, and, given its experimental nature, it was preferable to narrow down the scope of application, while making it practical. This choice does not invalidate a general nature of the research.

Trident Genesis is an open-source software development technology, which has been developed by Fielden Management Services Pty. Ltd (Australia). It aims to

[3]Trident Genesis Github page – https://github.com/fieldenms/tg

tackle the core problems of systems analysis and design that are often associated with building sophisticated transactional business information systems. TG fits well into the definition of domain-driven development, as it shares the common language of domain modeling, speaking in terms of domain entities and their relationships.

1.5 Qualitative research

In order to assess the extent to which the stated objectives were achieved the experimental component of this work is carried out by employing an approach of qualitative research. The applictation of qualitative reserach to the field of software engineering is discussed by Hazzan & Dubinsky [OH14].

The conducted experiments involve a focus group of select software engineers from the industry, who are practicing domain-driven development as their main software design approach, making them ideal candidates to test our main hypotheses. We use a questionnaire as a main data gathering tool.

Richard Pawson [Paw04]'s work is one great example of an application of qualitative research methods to the field of software engineering.

1.6 Structure

This work is structured in the following way:

Chapter 2 goes into depth about key concepts and provides the necessary background for the rest of the paper.

Chapter 3 discusses related work, comparing the approaches employed. Each approach is examined in great detail with its strengths and limitations outlined.

Chapter 4 provides a detailed description of the implementation. It discusses the developed algorithm step-by-step with attached illustrations and examples.

Chapter 5 describes the experiment. It includes a display of the questionnaire contents and answers of the participants.

Chapter 6 discusses future work that encompasses a general framework independent approach and draws on shortcomings of the implementation.

Chapter 2

Background

This chapter covers the background for the fundamental concepts in this work in order to provide a deeper understanding of the research area and prepare the reader for the discussion of related work and the explanation of implementation details. We start with the topic of conceptual modeling in the context of software systems. Then, we introduce the approach of *domain-driven design*, which is followed by the description of the Java programming language and its application to this research.

2.1 Conceptual modeling of software systems

Conceptual modeling is an integral part of software construction. In software engineering field the term "conceptual schema" is often associated with the world of relational database management systems (RDBMS). A conceptual schema describes the structure of a database from a high-level perspective. More specifically, it identifies the core concepts and classifies them into tables, which contain columns that stand for the attributes of a concept. A conceptual schema does not include any internal details about a database.

For example, in a banking system domain, the conceptual schema might describe concepts such as *banking accounts and transactions*, as well as their relationship. One common language for expressing conceptual schemas is UML. Consider the banking example illustrated in the following UML class diagram:

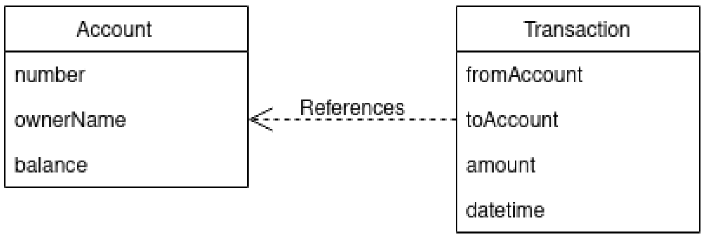


FIGURE 2.1: UML class diagram for a simplified banking domain

Here the conceptual schema consists of two domain entities. Both of them are characterized by specific attributes that describe their structure. And it is a common occurence that the underlying structure of an entity might undergo some kind of modification. For example, consider the diagram in Figure 2.2 that introduces a new attribute to the `Account` entity, namely, its creation date:

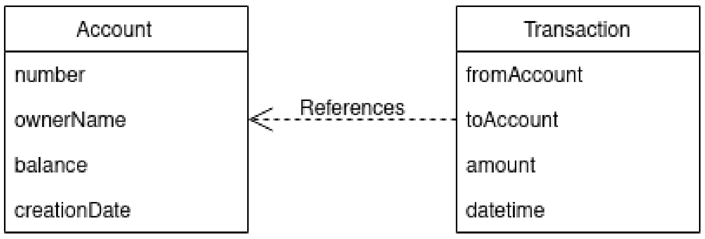


FIGURE 2.2: UML class diagram illustrating an additive change to the conceptual model

Modifications on a conceptual level, such as these, must be performed with caution, since they are fundamental in their nature. This means that every part of the system that interacts with a modified concept should be taken into account during verification. However, a simple additive change, as illustrated by the example, is relatively safe. Consider another example showing an existing attribute of `Account` being modified:

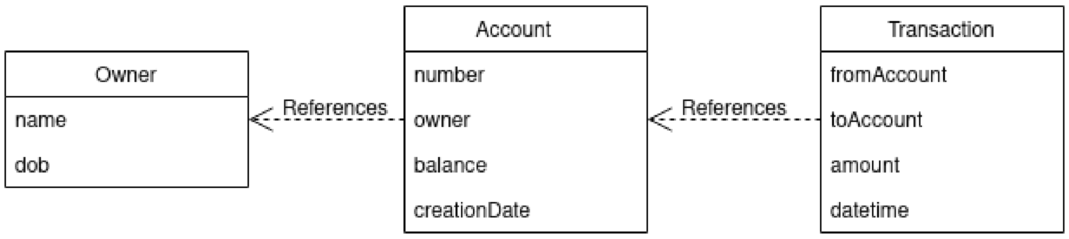


FIGURE 2.3: UML class diagram illustrating a breaking change to the conceptual model

The `ownerName` attribute of `Account` has been replaced by `owner`. The `Owner` concept has also been included in the diagram to provide the necessary context. This kind of change is often refered to as a *breaking change*, since it might "break" the system if any of its components were using the modified part. Given that prior to this change the `ownerName` attribute was a simple textual representation, any part of the system that was refering to this attribute must be modified accordingly. Ideally, after a modification has taken place, the system would be automatically validated to detect any errors that might have resulted from the modification. In our example it would be expected that the validation mechanism reports that the `Account` concept does not define an attribute `ownerName`, if it was used prior to the modification.

2.2 Domain-driven design and its terminology

Domain-driven design, a term coined by Eric Evans [Eva03], is an approach to conceptual modeling of a specific business domain. Its main aim is construction of a domain model, which translates to a conceptual model of the domain. At the core of domain-driven design one of the basic building blocks is an *entity*, that is, a domain object defined primarily by its identity. We use the term *entity* interchangeably with the term *concept* (assuming that the concept has an identity). Every entity is characterized (but not necessarily identified) by its *properties* in the same way that every concept is characterized by its *attributes*. The notion of domain-driven design is important since it enables an individual to think about systems design in terms of domain models.

2.3 Java programming language

Java [Jav] is a compiled, high-level, class-based, object-oriented programming language. In practice it is often used to implement domain-driven software systems, mapping domain objects to Java classes. Fields of a Java class are used to represent properties of an entity. For example, the domain from 2.1 could be modeled in the following way:

```
class Account {
    private int number;
    private String ownerName;
    private float balance;
}

class Transaction {
    private Account fromAccount;
    private Account toAccount;
    private float amount;
    private DateTime datetime;
}
```

LISTING 7: The domain illustrated in 2.1 modeled in Java.

2.4 Java reflection

The reflection capabilities of Java are briefly discussed in this section, since they are often mentioned alongside with the terms *metadata* and *metaprogramming*. In contrast with other compiled languages, such as C, Java is characterized by its sophisticated runtime mechanism that provides dynamic capabilities, such as reflection and code modification. Reflection, in particular, makes metaprogramming possible, which allows the program to use other programs, itself included, as data. For example, using reflection it is possible to obtain the type of a class field by its name:

```
class Person {
    private Animal pet;
}

Person.class.getDeclaredField("pet").getType(); // class Animal
```

Although this is a powerful programming technique that allows runtime type introspection to take place, it can not be used to treat the source code of a program as data at compile time. The runtime nature of reflection indicates that reflection operates on objects constructed from compiled code. This means that metadata obtained by using reflection exists only at runtime of the system, which is out of compiler's scope.

2.5 Java annotations

The Java Language Specification [JG15] defines a special construct – annotations – that provides data about a program that is not part of the program itself, in other words, it has no direct effect on the operation of code that is annotated. Annotations may be used to provide additional information to the compiler or to be interpreted during runtime of the program. We focus on the compile time processing of annotations. As an example, the `@Deprecated` annotation can be used to tell the compiler to generate warnings in places where the annotated element is used.

```
class Transformer {
    @Deprecated
    public static boolean isTransformable(Item item) {
        ...
    }
}

// warn: the method Transformer.isTransformable(Item) is deprecated
Transformer.~~isTransformale~~(someItem);
```

The annotated method can still be used as if there was no annotation, but the warning makes it clear to a software engineer that the method is no longer supported and its usage is discouraged.

2.6 Annotation processing

Annotation processing, as its name implies, is a mechanism for processing annotations in the source code at compile time. An important distinction from the previously mentioned concept of reflection is that the input of an annotation processor is an AST[1] constructed from the source code.

The `@Deprecated` annotation in the example above is one of the Java built-in annotations. It gets processed by a built-in annotation processor that is a part of the Java compiler.

The standard library of Java includes a common interface to all annotation processors, so software engineers can provide their own implementations, and instruct the compiler to use them. Apart from issuing compile time warnings and errors, annotation processing supports programmatic generation of new code. The following figure depicts the compilation process of javac[2].

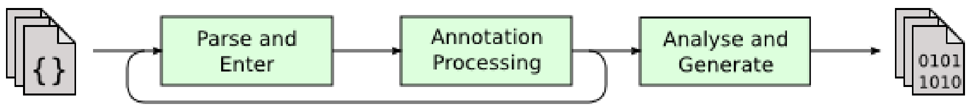


FIGURE 2.4: Compilation process of `javac`

Java compilation process can be broken down into the following steps:

1. The input source files are parsed to build a syntax tree.

[1]Abstract syntax tree is a tree-like data structure used by the compiler to represent the structure of program source code.

[2]Java compiler included in the JDK from Oracle

2. All appropriate annotation processors are run until no new files are generated.

3. The syntax tree is analyzed to generate byte-code.

It is important to mention that step 1 may differ across compiler implementations. The main difference lies in the input of a compiler. There is a notion of incremental compilation, which provides an ability to limit the number of source files that are passed as input to the compiler. As opposed to traditional compilation that requires the whole set of source files to be recompiled, an incremental compiler's input is limited to that portion of the program that was modified.

The built-in annotation processing API in Java[3] has a rich collection of types that model the source language itself, equipping programmers with a powerful abstraction to analyze the syntax tree.

[3] `javax.annotation.processing` - https://docs.oracle.com/javase/8/docs/api/javax/annotation/processing/package-summary.html

Chapter 3

Related work

This chapter contains an overview of existing approaches and examines the extent to which they might be used to achieve the objectives of this research. The scope of related work was not limited to the programming language used in the implementation. Given the dominance of Java in the field of software engineering, most of the related work presented in this chapter is naturally associated with it. Each approach is examined from the perspective of practicality, as well as in terms of the ability to reach the stated objectives.

3.1 C# nameof

C# is a programming language develop by Microsoft that is often put in comparison with Java, sharing a lot of fundamental principles. We inspect a particular expression that is a part of the C# language – the `nameof` expression [Mica] – that is used to obtain the name of a program element as a constant string. The general usage of this expression is demonstarted in the following example:

```
using System.Collections.Generic;

class Program {
    static void Main() {
        string s1 = nameof(Program);                     // "Program"
        string s2 = nameof(Program.InstanceMethod);      // "InstanceMethod"
        string s3 = nameof(System.Collections.Generic);  // "Generic"
        // Invalid
        string s4 = nameof(int);                 // Keywords not permitted
    }
    void InstanceMethod() { }
}
```

LISTING 8: Usage of `nameof` expression in C#.

Result of the `nameof` expression is evaluated at compile time. This means that `nameof(x)` is replaced by `"x"` in the compiled code, given that `x` is a legal input. Our interest is drawn, however, to the use of *property literals*[1] as string constants. And, indeed, `nameof` accepts class properties as input:

[1]A literal in programming is a notation used to for representing a fixed value in the source code. For example, 5 is a literal, "abc" is also a literal, but a variable name is not.

```
class Book {
    public string title { get; set; }
}
nameof(Book.title) // "title"
```

This is quite an effective way of using property literals as constant strings and it satisfies the compile time model correctness property. There are, however, several limitations to this approach, which are demonstrated below with provided examples:

1. The `nameof` expression cannot be applied to class members with restricted access, such as private members. Although we consider this a limitation, the example above, which makes use of C# Properties [Micb], is not constrained by it.

2. The nameof expression always returns a simple name of its input, that is, the resulting name is not fully-qualified. This was demonstrated in the first code snippet:

   ```
   string s3 = nameof(System.Collections.Generic);  // "Generic"
   ```

This limitation requires additional functionality to be implemented to concatenate the resulting string constants into a proper representation of the dot-notation.

```
class Book {
    public string title { get; set; }
    public Author author { get; set; };
}
class Author {
    public string fullName { get; set; }
}

// "author.fullName"
DotNotation.Of(nameof(Book.author), nameof(Book.author.fullName));

class DotNotation {
    public static string Of(params string[] names) {
        return String.Join(".", names);
    }
}
```

`Book.author.fullName` is used instead of simply `Author.fullName` to satisfy the compile time safety requirement. If the latter was used and the conceptual schema was changed leading to the type of `Book.author` not being `Author` anymore, but instead a value object or another entity that doesn't have the property `fullName`, a runtime error would occur at the moment of the dot-notation being used (to fetch some data from a database, for example). Consequently, this results in an unintuitive and overly verbose code.

3.2 Java nameof hack

While the Java programming language specification does not define property literals or an expression similar to that of C# `nameof`, there are 3rd party libraries [Art]

[Fra] that attempt to achieve the desired result by means of byte-code manipulation through the use of method references. What follows is a brief overview of the concepts of method references and reflection in Java.

3.2.1 Method references

Java has a special syntax for referring to a method of a class as if it was a lambda expression. Consider the following example:

```java
class Person {
    private String name;
    public String getName() {
        return this.name;
    }
}
```

This makes it possible to avoid writing a full lambda expression:

```java
map(person -> person.getName()); // lambda expression
map(Person::getName);            // method reference
```

3.2.2 Reflection and byte-code manipulation

Reflection is a feature in the Java programming language that allows a Java program to discover and manipulate information about itself in the runtime. More precisely, one can obtain information such as names of class members, their types, etc. In addition, it is possible to "intercept" a method of a class by using a *dynamic proxy*. In this case an interceptor would gain access to the information about the intercepted method, such as its name, parameter types, return type, etc.

3.2.3 The hack

It is possible to use a combination of method references and reflection to intercept the getter method call, such as `getName()` in the above example and map the method name to a field name, for which the getter was designed. As a result it would be possible to get the property name as a String in a compile time safe manner:

```java
nameOf(Person.class, Person::getName) // "name"
```

As this approach is merely mimicking the `nameof` expression of C#, it is subject to the same limitations. In addition, there is no way of chaining properties to construct a dot-notation in a compile time safe manner, since method references are semantically equivalent to lambda expressions.

3.3 Project Lombok

Project Lombok [Lomb] provides a handful of useful additions to the Java programming language in the form of annotations with the aim to reduce boilerplate code[2].

[2]Boilerplate is a term used for the parts of code that are repeated in multiple places with little to no variation

The specificity of this approach lies in the fact that lombok injects itself into the compilation phase to build on top of the source code being compiled, that is, lombok's annotations are used to replace repetitive pieces of code.

Project Lombok makes use of annotation processing for the sole purpose of identifying the supported annotations, i.e., as an entry point, and, unlike the original purpose of APT, does not generate any source files. Instead it uses the internal API of Java compiler (supports javac and ecj[3]) to manipulate the AST. The resulting AST is then analyzed and translated into bytecode. This process is illustrated by the following figure [nei11].

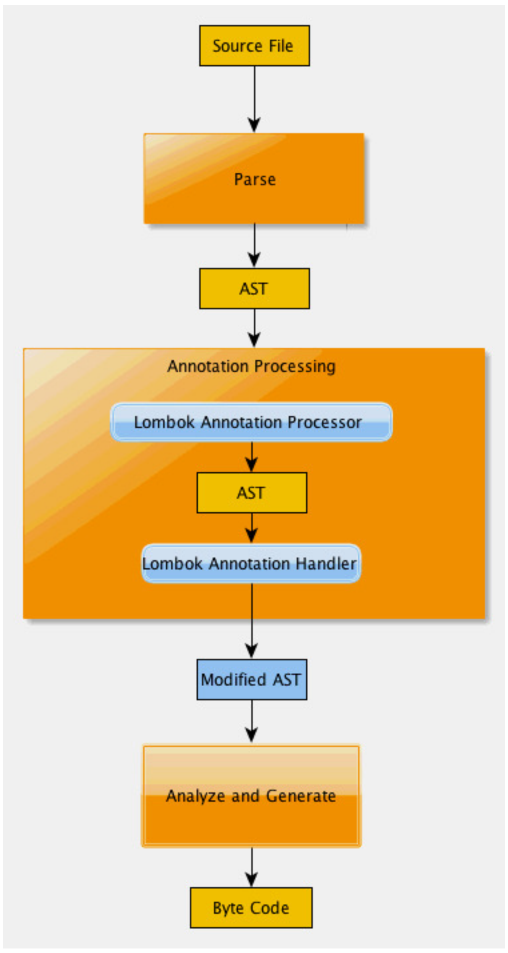


FIGURE 3.1: Lombok Annotation Processor modifies an AST

3.3.1 @FieldNameConstants annotation

One of the experimental features of Project Lombok, `@FieldNameConstants` [Loma] annotation is used to generate an inner type that contains a string constant representing a field's name for each field of the annotated type. This annotation is provided with some room for configuration, such as inclusion or exclusion of particular fields, capitalization of generated fields names, the name of the inner generated type, etc. The following listings demonstrate the benefits of using Lombok.

[3]Eclipse Compiler for Java

```java
@FieldNameConstants
public class Person {
    private String name;
    private int age;
    @FieldNameConstants.Exclude
    private String id;
}
```

LISTING 9: Java source code using Lombok.

```java
public class Person {
    private String name;
    private int age;
    private String id;

    public static final class Fields {
        public static final String name = "name";
        public static final String age = "age";
    }
}
```

LISTING 10: Java source code equivalent to 9, but without Lombok.

This approach, although effective for basic needs, is limited by its trivial nature:

- The generated entity graph is limited by a depth of a single level, as the generated fields are always of type `String`.
- Domain discoverability is limited by the lack of descriptiveness of the generated fields, since no javadoc accompanying a field is present.

However, the approach employed by Project Lombok is certainly a unique and inspiring piece of work to learn from.

3.4 Hibernate Metamodel Generator

Hibernate is an implementation for the JPA[4] [RB]. Hibernate Metamodel Generator [Fer10] is an annotation processing tool that is a part of the Hibernate ORM framework. It automates the generation of static entity meta-models used for typesafe Criteria queries as defined by the JPA 2. The queries benefit from the meta-models by being able to be constructed in a strongly-typed manner, thus avoiding the risks of type casting the result of a query. Consider the following example which shows a meta-model generated for the `Order` entity:

[4]Java Persistence API (renamed to Jakarta Persistence)

```
@Entity
public class Order {
    @Id
    private long id;

    @ManyToOne
    private Person customer;

    @OneToMany
    private Set<Item> items;

    private BigDecimal totalCost;
}
```

LISTING 11: Java class modeling the `Order` entity.

```
@StaticMetamodel(Order.class)
public abstract class Order_ {

    public static volatile SingularAttribute<Order, Long> id;
    public static volatile SetAttribute<Order, Item> items;
    public static volatile SingularAttribute<Order, BigDecimal> totalCost;
    public static volatile SingularAttribute<Order, Person> customer;

    public static final String ID = "id";
    public static final String ITEMS = "items";
    public static final String TOTAL_COST = "totalCost";
    public static final String CUSTOMER = "customer";

}
```

LISTING 12: Hibernate meta-model for the `Order` entity.

What stands out is that, apart from the property names, the generated meta-model also captures information about property types, providing an ability to use them in a compile time safe manner. Therefore, this approach could find more interesting uses than the previous ones.

The major limitation of Hibernate meta-models is that they are not designed for traversing an entity graph deeper than a single level. That is, the modeling technique of entity relationships is not advanced enough.

Chapter 4

Implementation details

This chapter describes the proposed implementation of a meta-model generation mechanism designed in the Java programming language. The source code of the implementation is hosted on GitHub[1].

The meta-model generation mechanism is implemented as an annotation processor, hereafter referred to as "domain model processor".

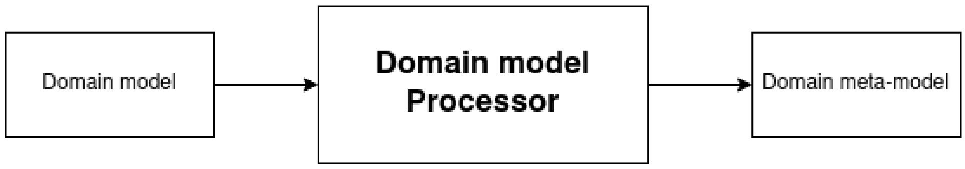


FIGURE 4.1: Domain model processor accepts a processing environment on input and produces generated sources on output

The annotation processor is initialized with a processing environment, by the compiler. The processing environment provides an AST that was obtained by parsing source files. In the incremental compilation environments, the processor benefits from the fact that it is possible to access the AST of a source file that was not necessarily a part of the compiler's input. The domain model processor performs semantic analysis of its input and obtains the domain model as a result.

We define the following domain evolution operations that are covered by the meta-model generation mechanism:

1. Creation of a domain entity.
2. Renaming of a domain entity.
3. Removal (deletion) of a domain entity.
4. Modification of a domain entity's structure that encompasses the following suboperations: renaming of a property, removal of a property, modification of a property's type or name, and creation of a new property.

Each of these operations leads to an according change in the conceptual schema. Therefore, it is important to be able to detect those changes (by means of semantic analysis) and reflect them in the meta-model.

[1]GitHub Issue - `https://github.com/fieldenms/tg/issues/849`.
Github Wiki page - `https://github.com/fieldenms/tg/wiki/MetaModels`.

4.1 Entity graph

A convenient and intuitive model for entities and their properties is a graph. The structure of an entity can be represented as a directed graph where each node is a type, with the *source* being the type of an entity itself. An arc (x, y) represents an association and can be read as "Entity x has a property of type y".

Figure 4.2 depicts an entity graph for `Person`, `User` and `Vehicle`. The labels attached to arcs are the corresponding names of those properties. All nodes are labeled with their type's name. The ones filled with blue are entity types, while those filled with white, which are always sinks, are non-entity types. An entity graph might contain a cycle.

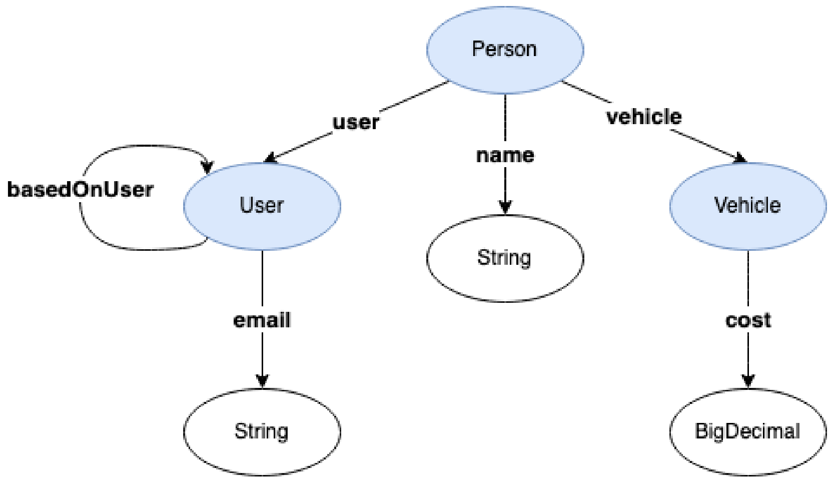


FIGURE 4.2: `Person` entity graph

This graph contains one cycle – $(User, User)$ at the `basedOnUser` property.

4.2 Meta-model generation algorithm

At the highest level the meta-model generation mechanism functions according to the following rules:

- For each class annotated with `@MapEntityTo` or `@DomainEntity` there will be a meta-model generated that captures all of its properties, that is, all fields annotated with `@IsProperty`.
- A meta-models collection class will be generated. This class is a container storing an instance of every active meta-model in a static field. In other words, it contains an entity graph for each domain entity.
- Whenever a domain entity is modified, the whole entity graph is considered for regeneration. That is, each node in the graph that represents a domain entity will have its metamodel regenerated. The renaming and deletion of an entity is also covered.
- If an entity should no longer be metamodeled, that is, it is either no longer annotated with the above mentioned annotations or deleted, then its meta-model is regenerated into an inactive one. Its entity graph is removed from the meta-models collection class.

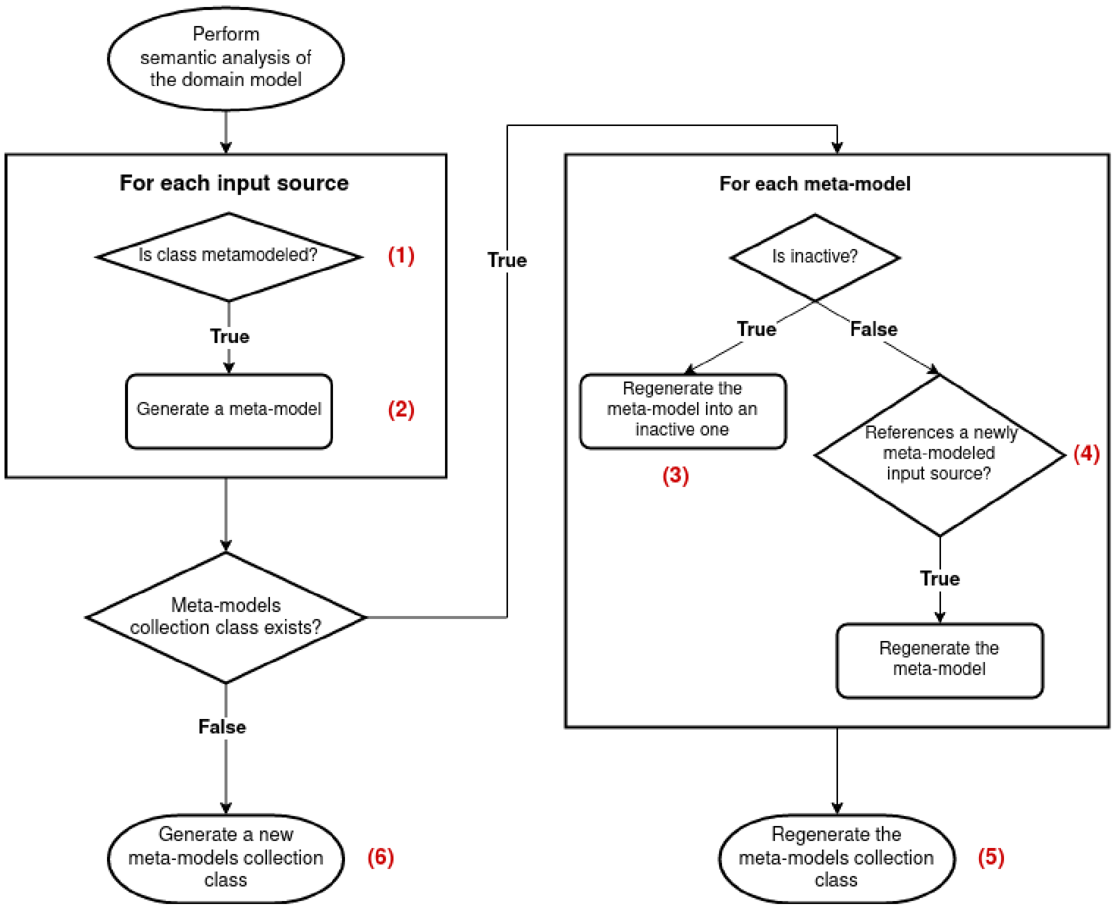

Perform semantic analysis of the domain model
For each input source
Is class metamodeled?
(1)
True
Generate a meta-model
(2)
Meta-models collection class exists?
False
True
Generate a new meta-models collection class
(6)
For each meta-model
Is inactive?
True
False
Regenerate the meta-model into an inactive one
(3)
References a newly meta-modeled input source?
(4)
True
Regenerate the meta-model
Regenerate the meta-models collection class
(5)

FIGURE 4.3: A high-level view of the meta-model generation algorithm

- **(1)** A class is considered to be metamodeled if it is annotated with `@MapEntityTo` or `@DomainEntity`.
- **(2)** This step is illustrated in detail by 4.4. See **(2)***
- **(3)** A meta-model is considered to be inactive if its underlying entity no longer qualifies for being metamodeled. An inactive meta-model is structured in such a way that it effectively becomes "useless". To achieve this in Java we generate an abstract empty class (with no members). The fact that the class is abstract means that it cannot be instantiated. The reasoning behind this choice was to overcome the limitations of an inability to support deletion of meta-models.
- **(4)** In case an entity becomes metamodeled, it is necessary to connect its meta-model to any existing ones, the underlying entities of which are adjacent in the entity graph. In other words, this step is responsible for connecting the entity graphs. This step is discussed in more detail below (4.5). See **(4)***.
- **(5)** The meta-models collection class is regenerated by removing fields with inactive meta-models and adding fields for newly generated meta-models if needed.
- **(6)** This step is equivalent to **(5)**, except that there are no inactive meta-models, since the collection class does not yet exist.

(2)*

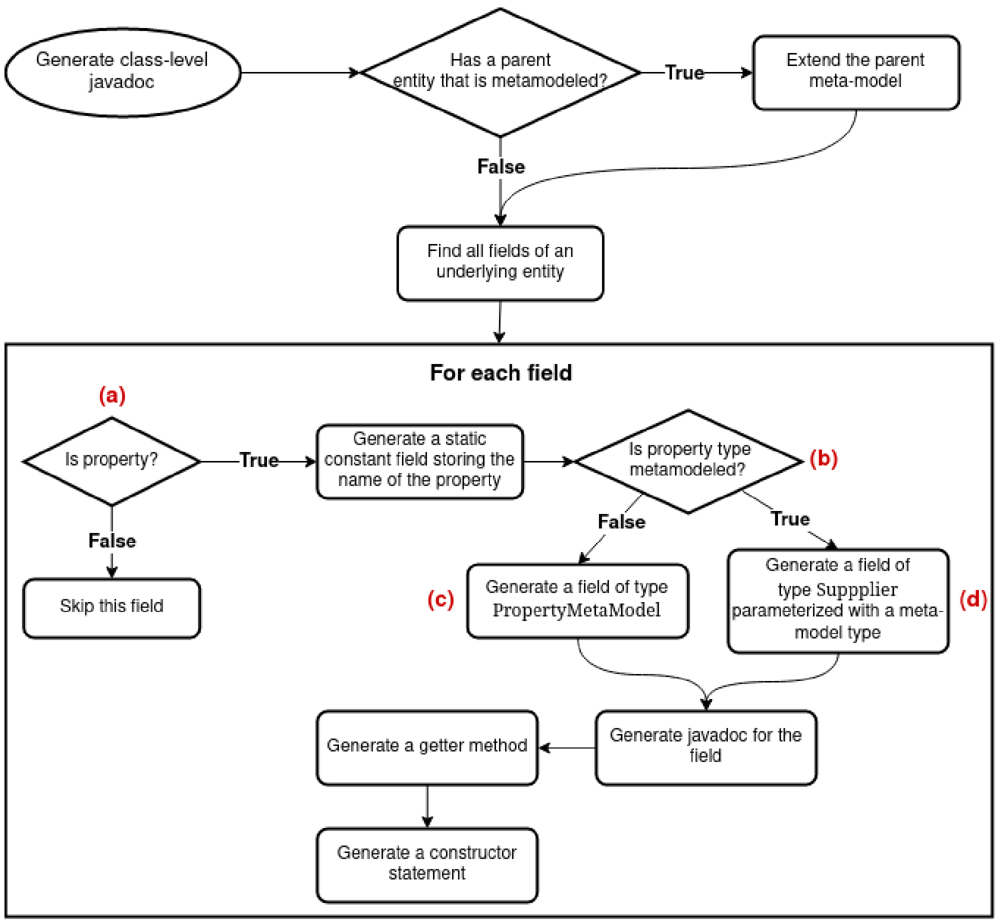

Generate class-level javadoc
Has a parent entity that is metamodeled?
True
Extend the parent meta-model
False
Find all fields of an underlying entity
For each field
(a)
Is property?
True
Generate a static constant field storing the name of the property
Is property type metamodeled?
(b)
False
True
False
Skip this field
(c)
Generate a field of type PropertyMetaModel
Generate a field of type Suppplier parameterized with a meta-model type
(d)
Generate a getter method
Generate javadoc for the field
Generate a constructor statement

FIGURE 4.4: The procedure of generating a meta-model

(a) Each field annotated with `@IsProperty` is considered to be a property of an entity.

(b) See **(1)** above.

(c) `PropertyMetaModel` class is used to represent any property that is a sink in the entity graph.

(d) `Supplier<T>` is a parameterized type used to represent any node in a graph that is not a sink, where `T` is a meta-model class. This particular type is used in order to achieve lazy computation of an entity node value, since an entity graph may contain a cycle.

(4)* Consider a situation where `Person` and `User` entities are metamodeled, while the `Vehicle` entity is not. Then the following entity meta-model graph exists:

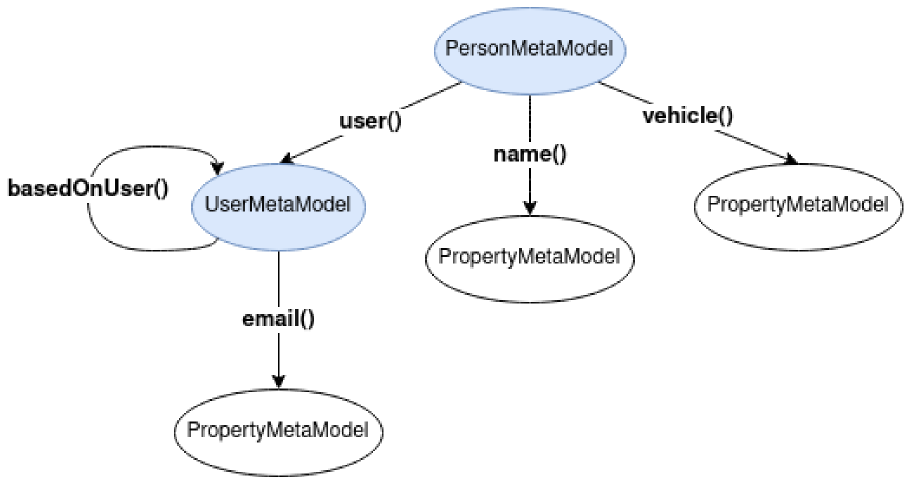


FIGURE 4.5: An entity meta-model graph where `Person` and `User` are metamodeled, but `Vehicle` is not

Then, if `Vehicle` becomes metamodeled, the arc $(PersonMetaModel, PropertyMetaModel)$ labeled `vehicle()` should be replaced by an arc $(PersonMetaModel, VehicleMetaModel)$.

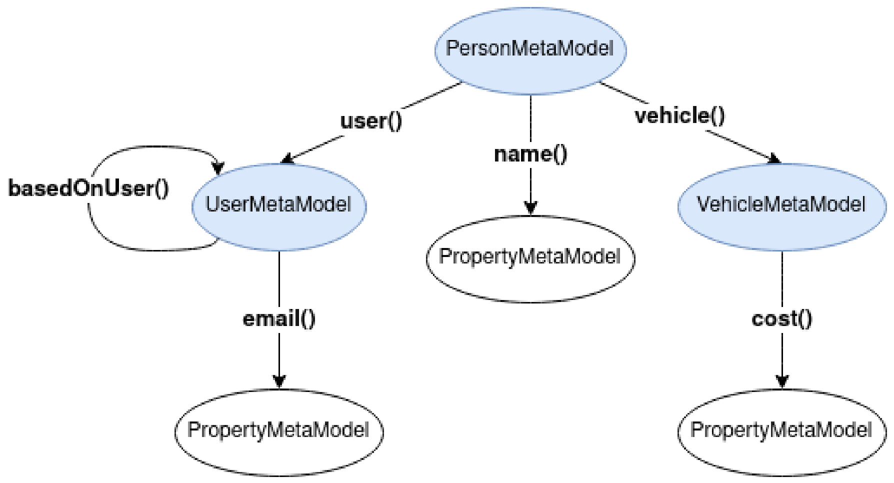


FIGURE 4.6: A meta-model graph for entity `Person` after `Vehicle` was metamodeled

This can be achieved only by traversing each entity graph in order to find the appropriate adjacent entity nodes ($(Person, Vehicle)$ in the example above).

4.3 Meta-model structure

The following UML class diagram illustrates the structure of the generated meta-models for entities from the example in 4.2.

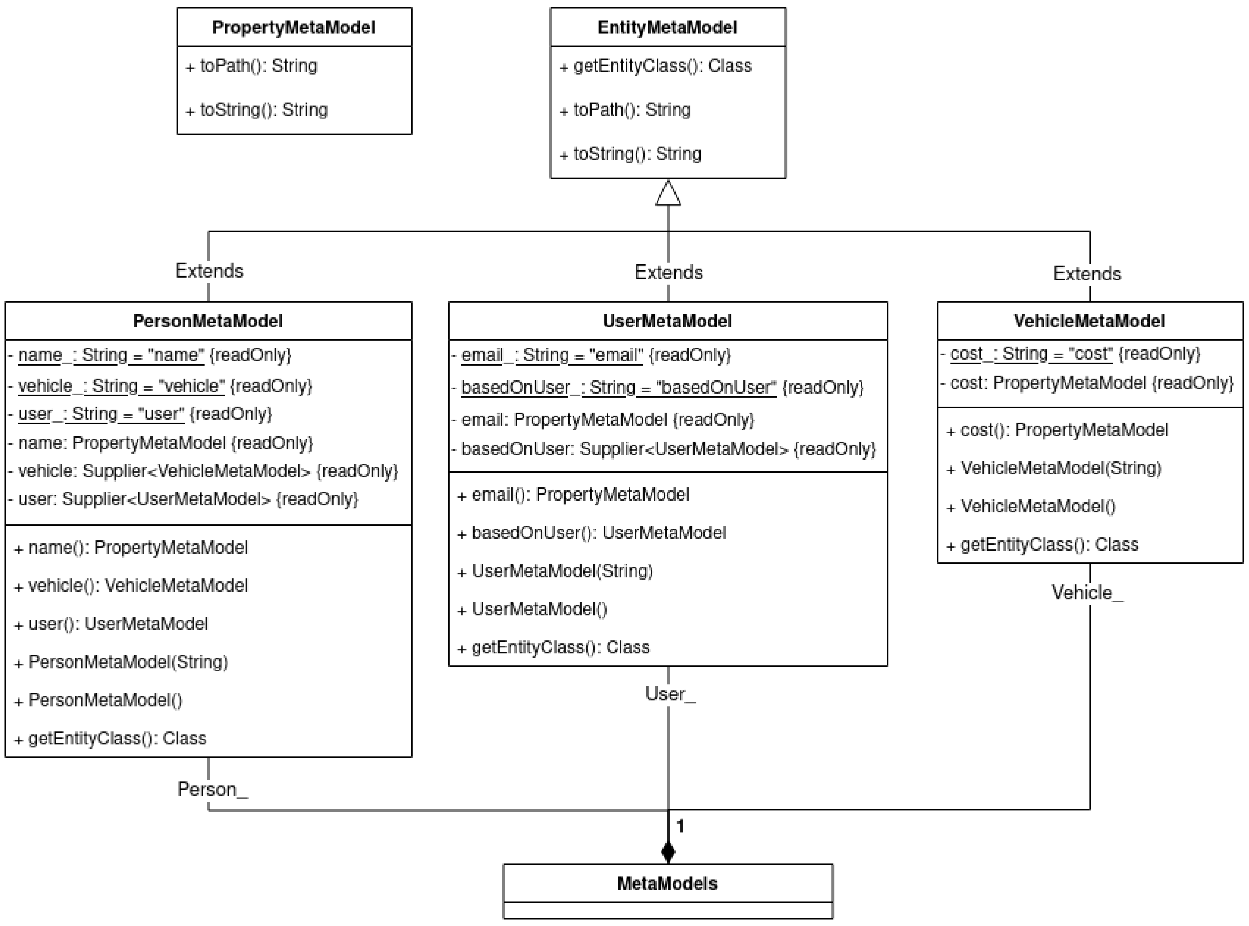

PropertyMetaModel
+ toPath(): String
+ toString(): String
EntityMetaModel
+ getEntityClass(): Class
+ toPath(): String
+ toString(): String
Extends
Extends
Extends
PersonMetaModel
- name_: String = "name" {readOnly}
- vehicle_: String = "vehicle" {readOnly}
- user_: String = "user" {readOnly}
- name: PropertyMetaModel {readOnly}
- vehicle: Supplier<VehicleMetaModel> {readOnly}
- user: Supplier<UserMetaModel> {readOnly}
+ name(): PropertyMetaModel
+ vehicle(): VehicleMetaModel
+ user(): UserMetaModel
+ PersonMetaModel(String)
+ PersonMetaModel()
+ getEntityClass(): Class
UserMetaModel
- email_: String = "email" {readOnly}
- basedOnUser_: String = "basedOnUser" {readOnly}
- email: PropertyMetaModel {readOnly}
- basedOnUser: Supplier<UserMetaModel> {readOnly}
+ email(): PropertyMetaModel
+ basedOnUser(): UserMetaModel
+ UserMetaModel(String)
+ UserMetaModel()
+ getEntityClass(): Class
VehicleMetaModel
- cost_: String = "cost" {readOnly}
- cost: PropertyMetaModel {readOnly}
+ cost(): PropertyMetaModel
+ VehicleMetaModel(String)
+ VehicleMetaModel()
+ getEntityClass(): Class
Vehicle_
User_
Person_
1
MetaModels

FIGURE 4.7: UML class diagram describing the generated meta-models for the domain illustrated in Figure 4.2

The actual `String` value representing the property dot-notation is accessed by calling the `toPath()` method (or the equivalent `toString()`). A meta-model also implements a `getEntityClass()` method that can be used to obtain the class of its underlying entity. Fields of the meta-models collection class (`MetaModels`) are named by appending the underscore to the name of the underlying entity in order to avoid conflicts that might be caused by static imports in Java (`Person` as type and `Person` as statically imported field). Methods of a meta-model that are used to traverse the entity graph contain additional information about the modeled properties in the form of javadoc as illustrated below:

```java
@IsProperty
@Title("Name", desc = "The name of this person")
@MaxLength(255)
private String name;
```

LISTING 13: An arbitrary non-entity type property (a sink node in the graph)

```
/**
* Title: Name
* Description: The name of this person
* Type: {@link String}
* {@literal @}{@link IsProperty}
* {@literal @}{@link MaxLength}(value = 255)
*/
public PropertyMetaModel name() {
    return this.name;
    }
```

LISTING 14: A property metamodeled after 13

```
@IsProperty
@Title("User", desc = "User associated with this person")
private User user;
```

LISTING 15: An arbitrary entity-type property

```
/**
 * Title: User
 * Description: User associated with this person
 * Type: {@link User}
 * Meta-model: {@link UserMetaModel}
 * {@literal @}{@link IsProperty}
 */
public UserMetaModel user() {
    return this.user.get();
}
```

LISTING 16: A property metamodeled after 15

4.4 Usage example

The following listing shows how a meta-model might be used:

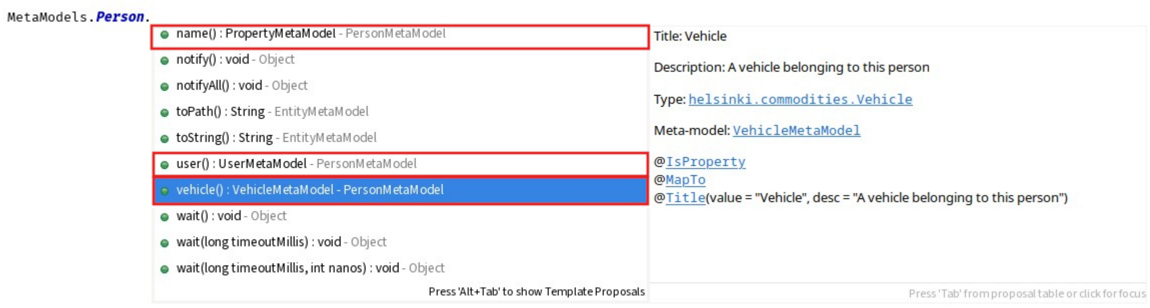
```
1   public class PersonFetcher {
2       static final PersonMetaModel person = MetaModels.Person_;
3
4       // a fetch model used to fetch data from a database
5       static final Fetch<Person> FETCH = fetch(Person.class).with(
6               // sink node
7               person.name(),                  // "name"
8
9               // entity node
10              person.user(),                  // "user"
11              person.user().basedOnUser(),    // "user.basedOnUser"
12              person.vehicle(),               // "vehicle"
13              person.vehicle().cost(),        // "vehicle.cost"
14
15              // source node
16              person                          // "this"
17              );
18  }
```

LISTING 17: Using the meta-model for entity `Person` to traverse its graph

Suppose that the conceptual schema has changed, resulting in the `Vehicle` entity no longer having property `cost`. Then, the following compilation error would occur at line 13:

```
// error: The method cost() is undefined for the type VehicleMetaModel
person.vehicle().cost(),
```

The true power of a meta-model is manifested in combination with code auto-completion:



```
MetaModels.Person.
● name() : PropertyMetaModel - PersonMetaModel
● notify() : void - Object
● notifyAll() : void - Object
● toPath() : String - EntityMetaModel
● toString() : String - EntityMetaModel
● user() : UserMetaModel - PersonMetaModel
● vehicle() : VehicleMetaModel - PersonMetaModel
● wait() : void - Object
● wait(long timeoutMillis) : void - Object
● wait(long timeoutMillis, int nanos) : void - Object
Press 'Alt+Tab' to show Template Proposals

Title: Vehicle

Description: A vehicle belonging to this person

Type: helsinki.commodities.Vehicle

Meta-model: VehicleMetaModel

@IsProperty
@MapTo
@Title(value = "Vehicle", desc = "A vehicle belonging to this person")

Press 'Tab' from proposal table or click for focus
```

FIGURE 4.8: Traversing the entity graph with the help of Eclipse IDE code auto-completion feature
Note: Property-methods are highlighted

Chapter 5

Evaluation

Evaluation of the developed technology was conducted in the form of an experiment. The motivation behind this choice was the desire to assess the extent to which all stated objectives were achieved. Although, the intuitive choice was to employ an approach of automated software testing, not all objectives could be effectively assessed in that manner. Domain discoverability, in particular, is better suited to evalutaion by opinion because of its subjective nature. Also, given rather uncommon software development technique of code generation, as well as the intricate dependencies between the generated code and source code, it was challenging to develop automated tests for the assessment of the other two objectives, namely, model consistency and evolvability. Therefore we use the experiment as the sole means of evaluation.

The focus group of the experiment was comprised of 7 software engineers working on several commercial projects built with TG at their core. The experiment had been conducted over a period of one week, at the end of which every participant was asked to fill out a questionnaire. Each individual was asked to agree/disagree with a handful of statements and optionally provide additional comments.

Despite the fact that the actual duration was shorter than originally planned, most answers convey enough information with only a few where several respondents share the opinion that more time is required. What follows is a display of responses, accompanied by selected comments that contributed most valuable insight.

5.1 Discoverability of the domain model

Figure 5.1 shows that all respondents recognize the improvements in domain discoverability. Several comments show an appreciation for the generated javadoc for entity properties, implying that it has led to an increase in the development pace.

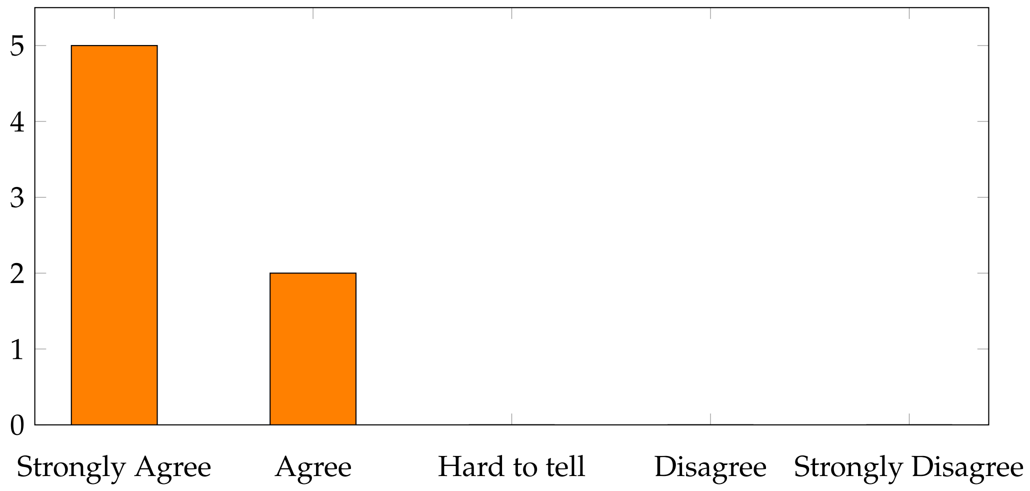


FIGURE 5.1: Responses to the statement: **The information provided by meta-models in the form of javadoc combined with the IDE's code auto-completion feature made domain discoverability more efficient.**

Figure 5.2 shows that the need for repetitive context switching was effectively eliminated with the introduction of meta-models. One participant commented that he was particulary annoyed in the past by the need for context switching. However, several other comments indicate that it was occasionally necessary to switch to the definition of an underlying entity to further discover additional information that is out of scope of the meta-model.

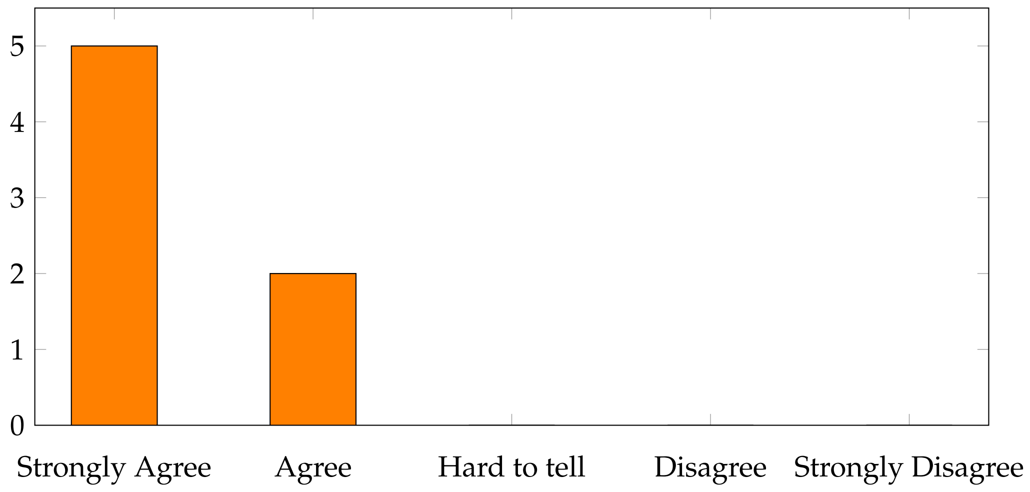


FIGURE 5.2: Responses to the statement: **When attempting to refer to a property of an entity using a meta-model, there was no need for context switching, i.e., opening an entity class and looking for the property definition.**

The opinions vary in regards to whether meta-model conveys enough information about the underlying domain entities, as illustrated by Figure 5.3. One comment suggested an idea that a meta-model could contain the database representation of its underlying entity (e.g. table name and columns data types).

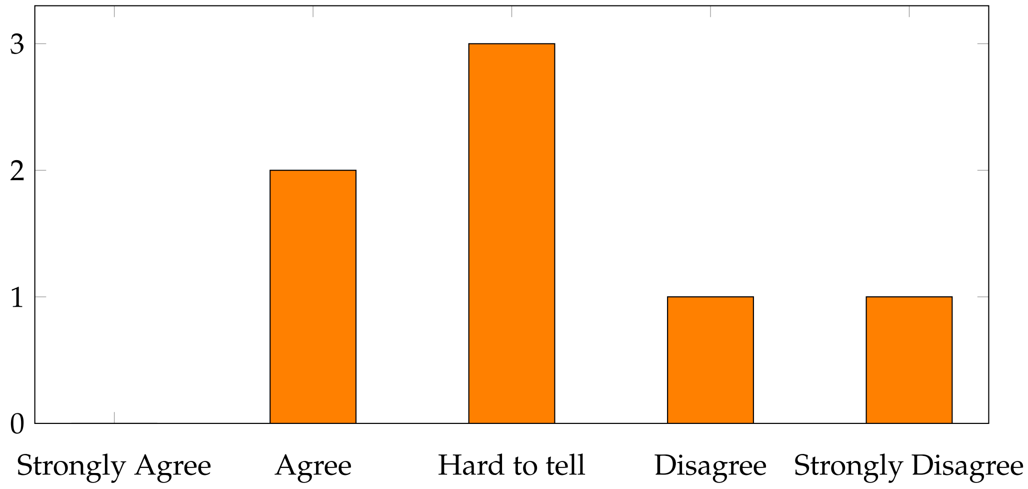


FIGURE 5.3: Responses to the statement: **The generated meta-models could contain more information about their underlying entities.**

5.2 Reliability

Figure 5.4 shows that all participants are in agreement about reliability of the meta-models. Several comments identified improvement areas for the meta-model. One participant suggested that it would be practical to to allow external properties (represented by String objects) to be inserted into the dot-notated path of a meta-model. Another participant identified a limitation of the meta-model in the fact that it can not be used in annotations, that is, as a constant value at compile time. We address these ideas in Chapter 6 in the discussion of future work.

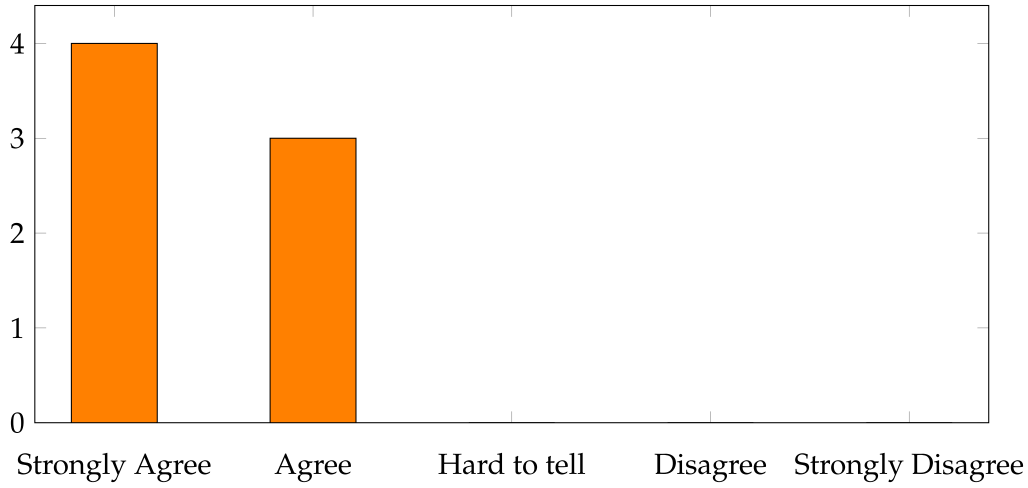


FIGURE 5.4: Responses to the statement: **Usage of the generated meta-models proved to be a reliable way of referencing properties of an entity, i.e., there were no occurrences of runtime errors.**

5.3 Evolvability

Figure 5.5 shows that most respondents acknowledge that system evolvability has improved with the introduction of meta-models.

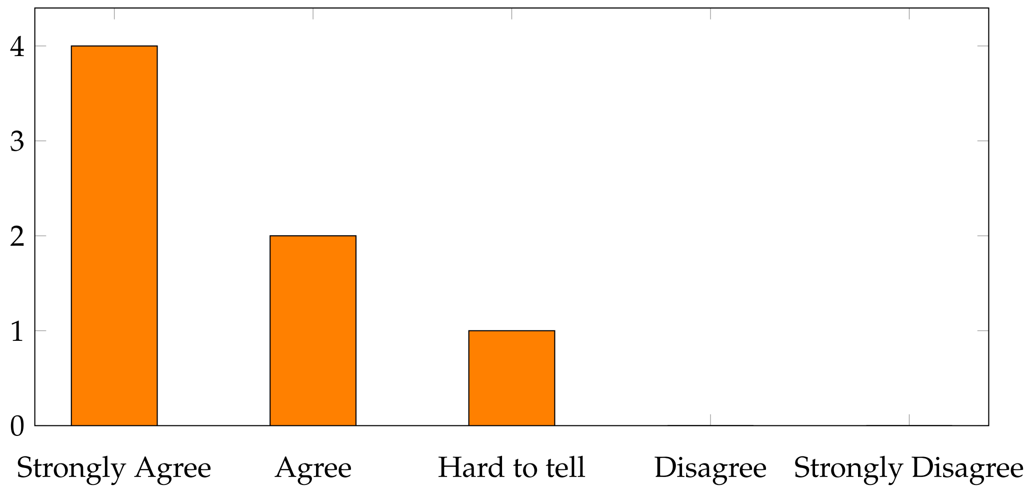


FIGURE 5.5: Responses to the statement: **Evolvability of a system against modifications to the conceptual model increased due to compile-time validation in places where the generated meta-models were referenced.**

5.4 Performance

Figure 5.6 shows that the impact on performance of an IDE was not of a noticable significance. Two participants commented that even for relatively large domains the generation process was very fast. One of the "Hard to tell" answers was followed by a comment that more time is needed to asssess this aspect.

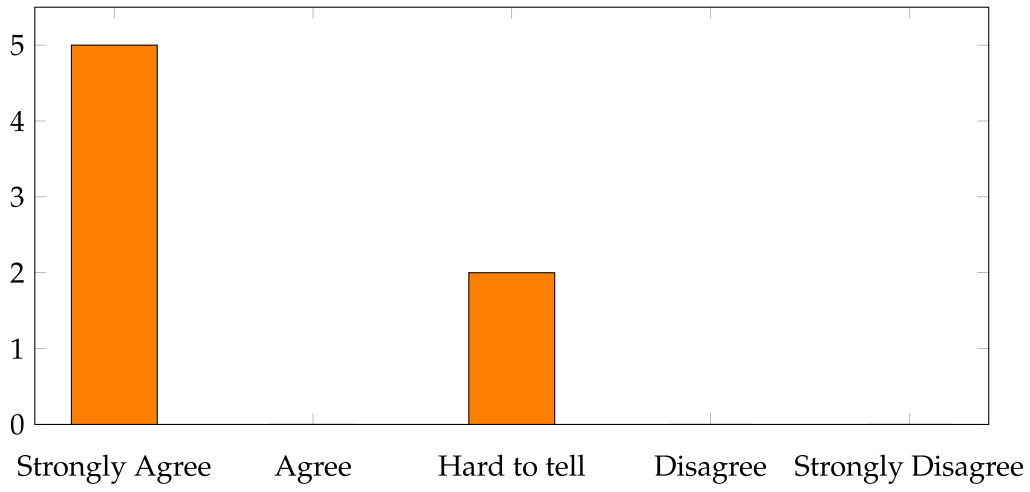


FIGURE 5.6: Responses to the statement: **Impact of the meta-model generation process on performance of the IDE during compilation has been insignificant to the development process.**

5.5 Correctness of the generation mechanism

Figure 5.7 shows that most participants agree that the generated meta-models were correctly reflecting the additive changes to the domain model. Several comments, however, state that more time is required for evaluation of this aspect, since addition of an entity is not a common occurence.

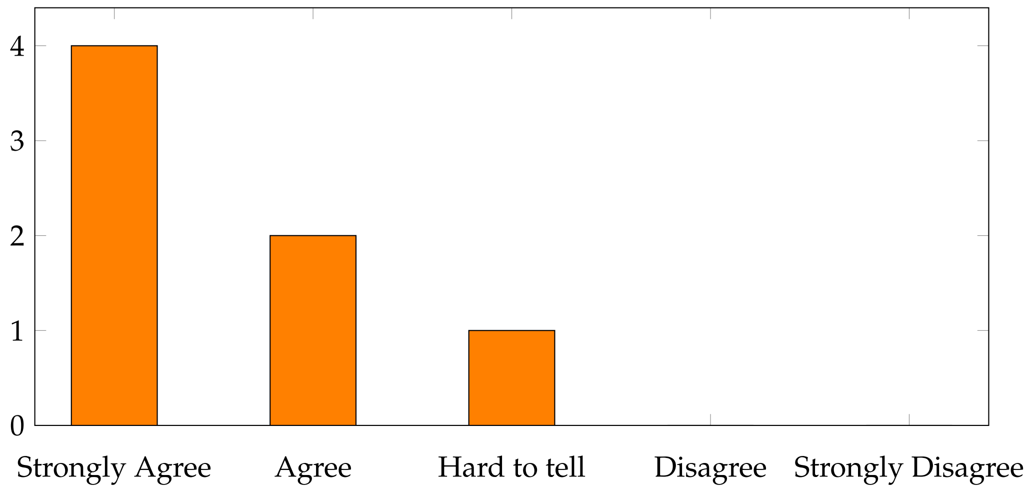


FIGURE 5.7: Responses to the statement: **The meta-model generation mechanism was always correctly generating meta-models for newly added entities.**

Figures 5.8 and 5.9 show that modifications of existing entities were successfully reflected in the generated meta-models. Both also share similar comments that express the need for more time for evaluaton.

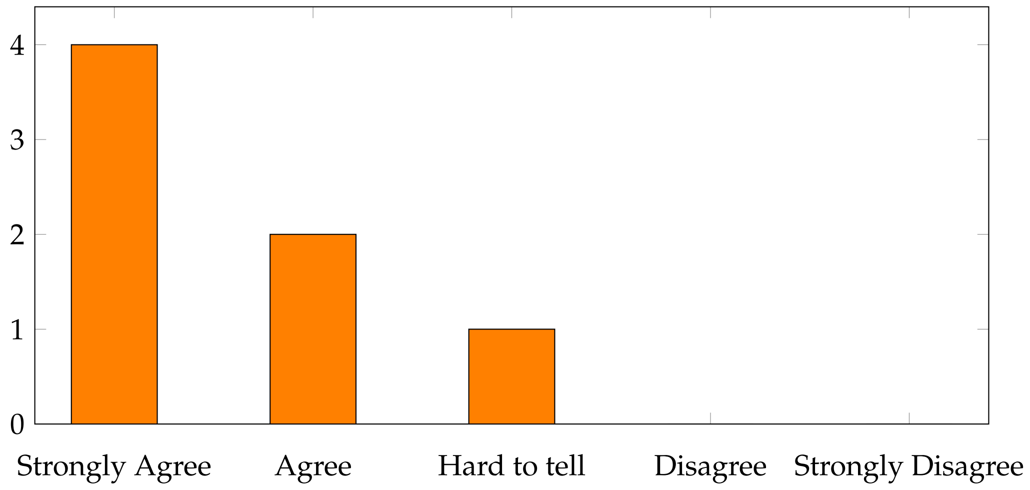


FIGURE 5.8: Responses to the statement: **The meta-model generation mechanism was acting correctly in response to the renaming / deletion of an entity.**

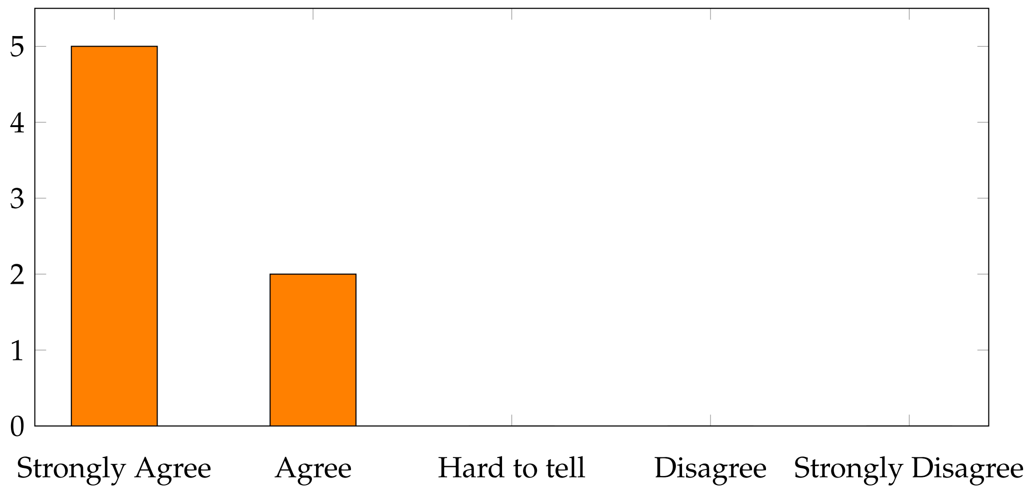


FIGURE 5.9: Responses to the statement: **The meta-model generation mechanism was always correctly adapting the latest modifications to the conceptual model.**

5.6 Intuitiveness and ease of use

Responses illustarted by Figure 5.10 indicate that all participants found meta-models easy to understand and use. One respondent identified a problem that he experienced while using the meta-model that arose from the fact that the metamodeled properties were mixed with other methods that are inherited by a meta-model class, thus making it difficult to distinguish them. This effect can be observed in Figure 4.8, where only property related methods are highlighted.

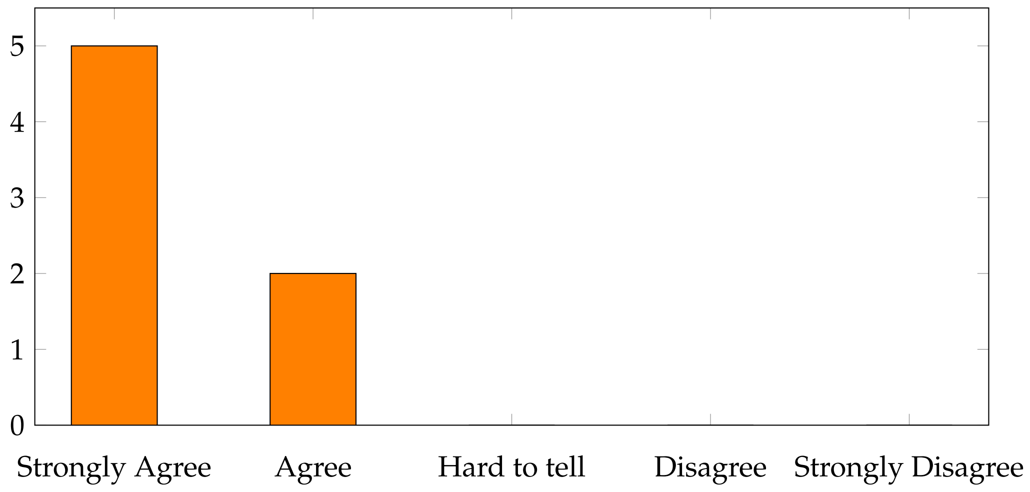


FIGURE 5.10: Responses to the statement: **Overall structure of the entity meta-model was easy to understand and intuitive in its use for referencing and chaining properties of an entity.**

Figure 5.11 shows that the majority of responses indicate an uncertainty in regards to improvements of meta-model structure. One of the respondents commented that it would be beneficial to broaden the scope of meta-models to cover Java types that are not a part of a domain (e.g. `BigDecimal`, `Integer`). This further shows that there is an evident need for metamodeling facilities in programming languages.

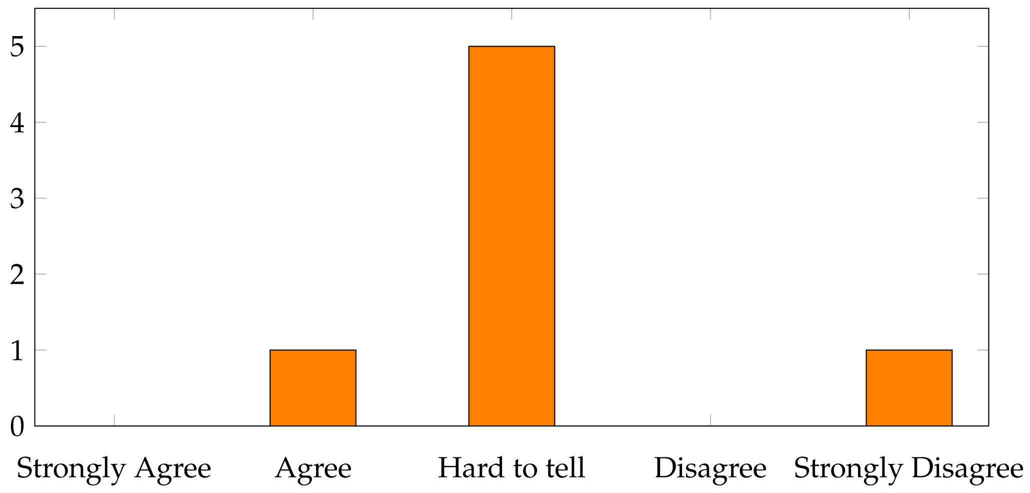


FIGURE 5.11: Responses to the statement: **The entity meta-model could be structured in a better way.**

Figure 5.12 shows that all participants felt positively about their experince of using meta-models. One of the respondents even admited that despite his reluctancy to changes he felt that the addition of meta-models was useful.

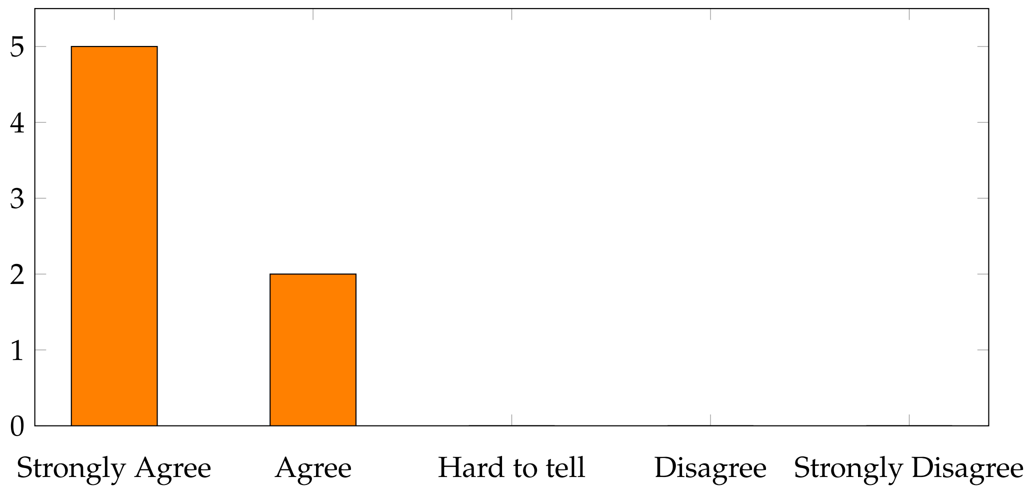


FIGURE 5.12: Responses to the statement: **Overall, I am rather satisfied with the experience of utilizing the meta-model generation tool.**

Chapter 6

Conclusion and future work

In this chapter, the research findings are reviewed against the original objectives set out in Chapter 1. This is followed by some suggestions for future work.

6.1 Review against the original objectives

Chapter 1 outlined three aspects of software construction to be improved: domain discoverability, system reliability and system evolvability.

In Chapter 1 we stated our hypothesis that the lack of metamodeling facilities on the level of programming language that provide design-time domain discoverability features is an accidental complexity involved in the process of software construction. It was shown that the consequences of this complexity limit the extent to which systems can be made reliable and software evolvable.

In Chapter 3 by examining related work we showed that the grounds for our hypothesis find their place in the software engineering field. It was found that the developers of Hibernate, a widely-used ORM framework in Java, were also aware of the issue at the core of our hypothesis. Their solution, while satisfying the needs of the framework, was deemed to be not advanced enough to facilitate domain discoverability.

Chapter 4 introduced an approach built around the concept of meta-model. It uses the Java annotation processing API to perform semantic analysis of the domain model in order to generate a meta-model of the domain.

Chapter 5 described the results obtained by evaluating the implementation. Our findings showed that when the approach was used in application to transactional business information systems it led to improvements in all three aspects, as was judged by the focus group of software engineers.

6.2 Future work

During the course of the research several potentially valuable areas of improvement have been identified. Some of them were brought to our attention during the experiment stage.

6.2.1 Integration with IDEs

Considering the significant role of IDEs in software engineering, the need for integration with the meta-model was identified as one of the most beneficial improvements. Firstly, it is desirable to enhance the refactoring capabilites of an IDE by instructing the underlying mechanism to take the references to meta-models into account during this process. This would reduce the time spent on the repetitive activity of modifying the entity graph paths in the source code. Secondly, it would be fitting if the methods of a meta-model class would benefit from syntax highlighting features of an IDE in the same way as the fields of a class do when they are referenced. This would result in better code readability.

6.2.2 Compile time constant values

As was mentioned by one of the participants of the experiment, the meta-model's nature does not allow software engineers to use it as a constant value at compile time, forcing them to revert to the usage of unreliable textual representation. We admit that our implementation of the meta-model is not able to manifest its true power at compile time due to the limits of Java programming language. However, this limitation can be partially overcome by providing a single level entity graph traversal capabilities at compile time, thus it is considered an attainable goal. Also, the need for traversing an entity graph deeper than a single level at compile time has not been identified yet.

6.2.3 Support for external metadata

During the experiment one participant invented a new use for the meta-model. He identified the need for insertion of external metadata into the constructed paths from traversing an entity graph. Using this approach would make it possible to combine the meta-model with other sources of metadata. We believe that this would further expand the metamodeling capabilites and recognize this as a feasible improvement.

6.2.4 General framework indepent approach

The idea of creating an abstraction for metamodeling capabilities opens the door to vast improvements in information systems design and software engineering in general. Such an abstraction could be designed in the form of an annotation processor with the core logic similar to that presented in this research. A framework independent implementation in Java is a task that can be accomplished by designing a general interface for a framework to conform to. The use of reflection is proposed to link the interface implementation to the abstraction (meta-model generation mechanism).

Bibliography

[Art] Yuriy Artamonov. Java alternative to nameOf operator. URL: https://github.com/strangeway-org/nameof.

[BM06] Peter Marks Ben Moseley. “Out of the Tar Pit”. In: (2006).

[Eva03] Eric Evans. Domain-Driven Design: Tacking the Complexity in the Heart of Software. Pearson Education (US), 2003. ISBN: 9780321125217.

[Fer10] Hardy Ferentschick. Hibernate JPA 2 Metamodel Generator. Mar. 10, 2010. URL: https://docs.jboss.org/hibernate/jpamodelgen/1.0/reference/en-US/html_single/.

[FPB86] Jr. Frederick P. Brooks. “No Silver Bullet — Essence and Accident in Software Engineering”. In: (1986).

[Fra] Florian Frankenberger. A Java library to programmatically return the name of fields similar to th URL: https://github.com/mobiuscode-de/nameof.

[Jav] Java. Oracle. URL: https://www.java.com.

[JG15] Guy Steele Gilad Bracha Alex Buckley James Gosling Bill Joy. The Java Language Specification. Feb. 13, 2015. URL: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf.

[Loma] Project Lombok. @FieldNameConstants. URL: https://notatube.blogspot.com/2010/12/project-lombok-creating-custom.html.

[Lomb] Project Lombok. URL: https://projectlombok.org/.

[Mica] Microsoft. nameof expression (C# reference). URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/nameof.

[Micb] Microsoft. Properties (C# Programming Guide). URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/properties.

[Nau85] Peter Naur. “Programming as Theory Building”. In: (1985).

[nei11] neildo. Project Lombok: Creating Custom Transformations. Jan. 6, 2011. URL: https://notatube.blogspot.com/2010/12/project-lombok-creating-custom.html.

[OH14] Yael Dubinsky Orit Hazzan. “Qualitative Research in Software Engineering”. In: (2014).

[Oli07] Antoni Olive. Conceptual Modeling of Information Systems. Springer, 2007. ISBN: 9780321125217.

[Paw04] Richard Pawson. “Naked objects”. PhD thesis. University of Dublin, Trinity College, 2004.

[RB] Ed Ort Rahul Biswas. The Java Persistence API. URL: https://www.oracle.com/technical-resources/articles/java/jpa.html.