Karlsruhe Institute of Technology

Software Design and Quality

Automatic Integration of Ecore Functionality into Java Code

Bachelor's Thesis of

Timur Sağlam

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Reviewer:	Prof. Dr. Ralf H. Reussner
Second reviewer:	Jun.-Prof. Dr.-Ing. Anne Koziolek
Advisor:	M.Sc. Heiko Klare
Second advisor:	Dr.-Ing. Michael Langhammer

19. December 2016 – 18. April 2017

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 10 April 2017

. . . .

(Timur Sağlam)

Abstract

Model-driven software development makes developing software faster and less complex by providing a higher abstraction layer than source code. However, since model-driven tools often generate the source code partially or completely from a model, one needs a suitable model to use model-driven approaches. The Eclipse Modeling Framework offers a meta-metamodel called Ecore, which can be used to create metamodels that can be utilized for model-driven software development.

With the Eclipse Modeling Framework, it is possible to generate Java code from Ecore metamodels. This generated code follows specific patterns. For example, it distinguishes between interface and implementation for every type. Many tools rely on these specific patterns of the generated code and therefore can only be used with Java code that contains these patterns, which usually is code generated from Ecore metamodels.

This thesis introduces an approach for the automatic integration of Ecore functionality into arbitrary Java code. This approach allows using Ecore-dependent tools on existing Java code. To accomplish the integration of Ecore functionality into Java code, this thesis develops a reverse engineering approach for the extraction of Ecore metamodels from Java code. At its core is a mapping from elements of the implicit Java metamodel to elements of the Ecore meta-metamodel. This extraction can also be used independently from the integration of Ecore functionality as a reverse engineering tool.

We provide a validation of our approach by integrating Ecore functionality into two Java projects and applying a model transformation on those projects, which was previously only possible with code generated from Ecore metamodels.

Zusammenfassung

Modellgetriebene Softwareentwicklung erlaubt schnelleres und vereinfachtes Entwickeln von Software durch das Bereitstellen einer höheren Abstraktionsebene als Code. Modellgetriebene Programme generieren Code oft teilweise oder vollständig mithilfe von Modellen. Aus diesem Grund benötigt man ein geeignetes Modell, um modellgetriebene Prozesse verwenden zu können. Das Eclipse Modeling Framework bietet das sogenannte Ecore Metametamodell, welches das Erstellen von Metamodellen ermöglicht, die für modellgetriebene Softwareentwicklung genutzt werden können.

Mithilfe des Eclipse Modeling Frameworks ist es möglich, Java Code zu generieren. Dieser generierte Code folgt spezifischen Mustern. Zum Beispiel unterscheidet er für jede Klasse zwischen Schnittstelle und Implementierung. Viele Programme sind auf diese spezifischen Muster angewiesen und können deshalb nur mit Java Code verwendet werden, welcher diese Muster enthält. Das ist üblicherweise Code, der mithilfe eines Ecore Metamodells generiert wurde.

In dieser Arbeit stellen wir einen Ansatz für die automatische Integration von Ecore Funktionalität in Java Code vor. Dieser Ansatz ermöglicht die Verwendung von Ecore-abhängigen Programmen mit existierendem Java Code. Um die Integration von Ecore Funktionalität zu ermöglichen, entwickelt diese Arbeit einen Ansatz für die Extraktion von Ecore Metamodellen aus Java Code. In ihrem Zentrum steht eine Abbildung von Elementen des impliziten Java Metamodells auf Elemente des Ecore Metametamodells. Diese Extraktion kann unabhängig von der Integration von Ecore Funktionalität als Reverse Engineering Programm verwendet werden.

Wir erbringen einen Nachweis für die Korrektheit unseres Ansatzes, indem wir Ecore Funktionalität in zwei Java Projekte integrieren und auf diese dann eine Modelltransformation anwenden, was zuvor nur auf Code, der aus Ecore Metamodellen generiert wurde, möglich war.

Contents

List of Figures

List of Tables

1. Introduction

Model-driven software development makes developing software faster and less complex by providing a higher abstraction layer than source code. It is also beneficial for the software quality [29]. However, since model-driven tools generate the source code partially or completely from a model, one needs a suitable model to utilize model-driven approaches. The Eclipse Modeling Framework offers an Essential Meta Object Facility (EMOF) compatible meta-metamodel called Ecore, which can be used to create metamodels that can be utilized for model-driven software development [31].

With the Eclipse Modeling Framework it is possible to generate Java code from Ecore metamodels. This generated code follows specific patterns. For example, it distinguishes between interface and implementation for every type. It also contains additional functionality like an object persistence management mechanism or a model change notifier system similar to the observer design pattern (see Chapter 2.4.1 in [31]). Many tools rely on these specific patterns of the generated code and therefore can only be used with Java code that contains these patterns, which usually is code generated from Ecore metamodels. Examples of such tools are model transformation languages like QVT [13] or ATL [16], consistency-preservation approaches like Vitruvius [20, 21], and editor frameworks like Sirius [36].

These tools, which rely on the Ecore functionality, cannot be used with arbitrary Java code. It is possible to model arbitrary Java code manually as an Ecore metamodel, generate the model code from that metamodel and then copy the implementation details into the model code. After that, the tools can be used with the model code. However, this process is not only very time-consuming but also API-breaking. A better idea would be to integrate the previously mentioned Ecore functionality into the existing Java code. This would allow the use of such tools on the Java code. This thesis introduces an approach for the automatic integration of Ecore functionality into arbitrary Java code. We call this approach *Ecorification* of Java code. One reason why there is a need for this approach is that software systems often are in use for a long time. As a result, many software projects were developed before model-driven software development techniques. This thesis could allow the developers of these legacy software systems to switch to model-driven software development.

1.1. Ecorification of Java Code

The Java code *Ecorification* has the goal to integrate Ecore functionality into Java code while preserving all of its original functionality. Preserving the original functionality of the Java code requires retaining the interfaces which are offered by the modules of the code and also retaining all internal functionality of the code. That means the product of the

Java code *Ecorification* is the original code, enriched with the desired Ecore functionality. This code then can be used with Ecore-dependent tools. At the current time, there are no existing approaches known to the author that allow the integration of Ecore functionality into Java code.

The basic idea of the *Ecorification* is to find an Ecore representation of the Java code. This representation is used to integrate its Ecore functionality into the Java code. The Ecore representation is obtained by creating an Ecore metamodel that represents the Java code as closely as possible. We use the Eclipse Modeling Framework to generate Ecore model code from the Ecore metamodel. That generated Ecore model code has similarities with the original Java code because we specifically created the Ecore metamodel as similar as possible to the Java code. In contrast, the model code also shows some differences when compared to the original Java code, because of the specific patterns of Ecore model code in general. As a result, the model code contains the desired Ecore functionality, which is its most important feature. We now use the similarities of the original Java code and the generated model code to interlace both codes. This can be achieved by utilizing the separation of interface and implementation to mount the original code into the super relation hierarchy of the model code. The combination of both codes then contains the implementation details of the original code and the Ecore functionality of the model code. This way, the *Ecorification* of Java code allows the integration of Ecore functionality.

The *Ecorification* of Java code was prototypically implemented as an Eclipse plug-in and serves as a proof of concept. The extraction implementation integrates Ecore functionality into the Java code of Java projects in the Eclipse workspace. The Java code *Ecorification* was validated through performing a QVTO transformation [13] on Java code, into which Ecore functionality was integrated through the *Ecorification* of Java code. Because QVTO transformations cannot be conducted on regular Java code, this validation proofed that this thesis was able to successfully integrate Ecore functionality into Java code.

1.2. Ecore Metamodel Extraction

Because the Java code *Ecorification* relies on an Ecore representation of the Java code, there is a need for a reverse engineering process that can automatically generate such an Ecore representation from any arbitrary Java code. The Eclipse Modeling Framework does not offer such a process. There are many existing approaches for reverse engineering models or metamodels from code. Most of them extract UML models from object-oriented code. At the current time, there are no approaches known to the author that could extract Ecore metamodels from object-oriented code.

As a result, this thesis proposes an approach for reverse engineering Ecore metamodels from Java code. This approach allows generating Ecore metamodels from any arbitrary Java code. These extracted metamodels can be used to generate model code. At the core of the approach is a mapping from elements of the implicit Java metamodel to elements of the Ecore meta-metamodel. The implicit Java metamodel is a metamodel, which the Java language implicitly defines because Java code follows a structure defined by the Java Language Specification [12]. This mapping defines how Ecore metamodel elements are extracted from Java code. We define the mapping for a higher metalevel to allow its application to any instances of the metalevel. For example, the mapping from the Java class to the Ecore metaclass *EClass* can be applied for any class.

The Ecore meta-metamodel is a simplified subset of UML. Therefore, the Ecore meta-metamodel is very similar to an UML class diagram (see Chapter 2.3.1 and 2.6.1 in [31]). However, the Ecore meta-metamodel has also its specific characteristics, which makes it in many ways fundamentally different. Because of this relation to UML, the Ecore meta-metamodel and the implicit Java metamodel are very similar, but not perfectly identical. That means there is no perfect mapping between the Ecore meta-metamodel elements and the implicit Java metamodel. As a result, not all features of Java code can be extracted, but the number of features that can be extracted allows the extraction of metamodels that accurately represent the Java code.

While the Ecore metamodel extraction approach was primarily developed for the Java code *Ecorification*, there are other applications for the Extraction of Ecore metamodels from Java code. The process of manually creating an Ecore model can be very time-consuming. It would be beneficial to be able to generate Ecore metamodels from existing software projects automatically. This is one example where the Ecore metamodel extraction can save much time by automating the process of creating an Ecore metamodel for an existing project.

The Ecore metamodel extraction was implemented as an Eclipse plug-in separately from the Java code *Ecorification*. While the *Ecorification* was only implemented as a proof of concept, the extraction implementation covers almost all features of the concept. While the extraction approach was primarily developed for the Java code *Ecorification*, it can also be independently used as a reverse engineering tool. The Ecore metamodel extraction was validated using three indicators: The validity of the extracted metamodel according to EMF, the ability to generate Ecore model code from the extracted metamodel, and indication through examined model properties. The examined model properties were compared to the correlating properties of the Java code used for the extraction. We tested the extraction for a custom designed test project and two larger, already existing projects. The Java code *Ecorification* was validated through performing a QVTO transformation [13] on Java code, into which Ecore functionality was integrated through the *Ecorification* of Java code. Because QVTO transformations can only be conducted on Ecore-dependent code, this validation proved that we were able to successfully integrate Ecore functionality into Java code.

1.3. Structure of this Thesis

Since the Ecore metamodel extraction can be used separately from the Java code *Ecorification*, we separate the Ecore metamodel extraction and the *Ecorification* of Java code throughout this thesis. Additionally, the thesis separates the theoretical concepts from their prototypical implementations. Chapter 2 introduces the foundations of this thesis. Next, Chapter 3 defines a running example which is used throughout the thesis to illustrate the different concepts on a simple level. This chapter also gives examples how one can understand the Ecore metamodel extraction and the *Ecorification* of Java code. Chapter 4 explains the concept of the Ecore metamodel extraction. It introduces the mapping from elements of the implicit Java metamodel to elements of the Ecore meta-metamodel and then elucidates how the different features of the Java language are extracted. Then, Chapter 5 explains the concept of the Java code *Ecorification* which uses the Ecore metamodel extraction to integrate Ecore functionality into Java code. It also discusses the problems which come with the *Ecorification* and suggests how they can be solved. Next, Chapter 6 covers the implementation of the Ecore metamodel extraction and the *Ecorification* of Java code. While the Ecore metamodel extraction implementation includes many features of the concept, the Java code *Ecorification* implementation is more a prototypical proof of concept. Chapter 7 explains how the concept of this thesis has been validated and what the difficulties of the validation are. The related work of this thesis and the differences to this approach is outlined in Chapter 8. Chapter 9 gives an overview of the unresolved problems of this thesis and possible future work. Last, Chapter 10 gives a summary of this thesis and its concepts.

2. Foundations

This chapter introduces the foundations on which this thesis is based on. First, the term model-driven software development is explained in Section 2.1. Second, the terms metamodeling, metamodels and meta-metamodel are introduced in 2.2. Third, the process of reverse engineering is outlined in Section 2.3. Fourth, the programming language Java is explained in Section 2.4. Then, the software project Eclipse is introduced in Section 2.5. After that, the Eclipse Modeling Framework is outlined in Section 2.6. The Ecore meta-metamodel is explained in Section 2.6.1, and the Ecore model code is explained in Section 2.6.2. Last, the term Ecore functionality is defined in Section 2.6.3.

2.1. Model-Driven Software Development

Model-driven software development is the application of model-driven engineering on software development. Its goals are reducing the complexity of developing software through working on a higher abstraction layer, increasing the development speed through automation, increasing the reusability, and better maintainability through redundancy avoidance. Model-driven software development is closely related to the Object Management Group's Model-Driven Architecture (MDA).

A model in the context of model-driven software development is an abstract representation of a system's structure. While in classic software development models are used to assist during the development process, model-driven software development makes models the primary tool for developing software. Model-driven software development uses models as the central element of the development. For example, models are used to automatically generate code, while classic software development usually uses models to generate a projection of the source code [29]

2.2. Metamodeling

Metamodeling is a very important aspect of model-driven software development. Metamodeling is used for the construction of domain-specific modeling languages, model validation, model-to-model transformations, code generation and modeling tool integration. Metamodeling describes the design and use of metamodels.

A metamodel is an abstract description of a model, and therefore the model of a model. It can be seen as a blueprint for specific models. That means that every model is an instance of a metamodel. Meta-metamodels are models of metamodels. The same description of metamodels can be used for meta-metamodels, only that meta-metamodels describe metamodels instead of models. Metamodels are one metalevel above models, and meta-

metamodels are one above metamodels. When talking about metalevels, a metalevel is always an instance of the metalevel above and describes the metalevel below.

While this layering could go on indefinitely, the Object Management Group defines four metalevels: Instances, models, metamodels, and meta-metamodels. The Object Management Group also defines an own metamodel: The meta object facility (MOF). There is no metalevel above the meta object facility. It rather defines itself. The Essential MOF or EMOF is a subset of the meta object facility. It allows the simplified creation of metamodels. The unified modeling language (UML), a well-known graphical modeling language developed by the Object Management Group, is an instance or application of the meta object facility [29].

2.3. Reverse Engineering

Reverse engineering in the context of software development describes either the process of extracting source code from binary files or the process of extracting models from existing systems. For this work, the second definition is relevant. Models are very useful in software development because they give a more abstract view on the system's code. They make it easy to understand complex structures and behavior. This gives the need for methods to extract models from existing source code [33, 10, 4].

Reverse engineering tries to extract as much information as possible from existing systems and represent them in a more abstract model. A common example is the reverse engineering of class diagrams from object-oriented code. An example of the application of reverse engineering is software maintenance. During software maintenance, reverse engineering helps to identify and understand the components and relations of a system [33, 10, 34].

2.4. Java

The Java language is an object-oriented general-purpose programming language. It first appeared in 1995 and was originally developed by James Gosling at Sun Microsystems, which is now owned by Oracle Corporation. Java code is compiled to Java bytecode, which is used by the Java Virtual Machine, an abstract computing machine, to run the Java code platform-independent. The goal of the language is to allow the development of "secure, high performance, and robust applications on multiple platforms in heterogeneous, distributed networks" (see Chapter 1.2 in GoslingMcGilton1995). The Java programming language sets out to be architecture neutral, portable, and dynamically adaptable [1, 11].

The syntax of the Java language is defined by the Java Language Specification. The Java Language Specification refers to its specific chapters and sections as paragraphs. This thesis refers to the Java Language Specification as JLS. Whenever JLS paragraphs are cited by this thesis, we specifically refer to the Java SE 8 Edition of the Java Language Specification (see [12]).

While the Java language does not explicitly define a model, the Java Language Specification can be interpreted as a textual representation of a metamodel. This metamodel is implicitly defined by the language Java. In this thesis, we refer to this metamodel as the implicit Java metamodel. This implicit Java metamodel is used by the Ecore metamodel extraction to define a mapping between the Ecore meta-metamodels and the Java language.

2.5. Eclipse

Eclipse is an open source software project which provides a highly integrated tool platform. The non-profit organization Eclipse Foundation leads the development of the Eclipse project. Eclipse is divided into several projects. The Eclipse Project develops the Eclipse IDE, an integrated development environment for Java. It contains four subprojects: Equinox, the Platform, the Java Development Tools and the Plug-in Development Environment. The Platform and Equinox are the core components of Eclipse and are therefore often confused with Eclipse itself. The Platform is a framework for developing Integrated Development Environments, while Equinox provides the component model, on which Eclipse is based. The Java Development Tools (JDT) is a Java development environment. It can be used to develop Java programs for Eclipse. Moreover, it is used to develop Eclipse itself. The Java Development Tools can be divided into three components: The Java Model, the Abstract Syntax Tree, and the Search Engine (see Chapter 1.1 and 1.2 in [31]).

The Java Model allows representing Java projects in a tree structure. It does not contain much information about the source code of a project itself but offers a fault-tolerant and lightweight representation of a Java project. The Java model can contain unresolved information. The Java Abstract Syntax Tree (AST) is a precise tree representation of Java source code. It is not as lightweight as the Java Model, but offers more functionality and does not contain unresolved information. Its API allows to create, modify and read Java source code. Moreover, it allows the Eclipse IDE to jump to declarations or to detect the declaration and all the references to a local variable. The Search Engine allows searching Java projects in the Eclipse IDE workspace for Java elements. It is possible to set a scope to search for specific elements and search for specific patterns inside Java elements (see [15, 22] and Chapter 1.1 in [31]).

2.6. Eclipse Modeling Framework

Another project is the Eclipse Modeling Project. Its task is the evolution of model-based development technologies. The core of the Eclipse Modeling Project is the Eclipse Modeling Framework (EMF), which adds modeling functionality to the Eclipse platform. The Eclipse Modeling Framework is an open source Java framework for modeling and code generation. It combines the Java language, XML, and UML in a high-level representation.

At its core, it uses an EMOF compatible meta-metamodel called Ecore (see Chapter 2 and 2.1 in [31]). The Ecore meta-metamodel is further explained in Section 2.6.1. EMF uses the Ecore meta-metamodel to represent other metamodels. Ecore metamodels can be created with a graphical EMF editor, annotated Java Code, XMI, an XML schema, UML and the Ecore API. The Ecore API allows programmatically creating, loading and saving Ecore metamodels. Eclipse offers two views for Ecore models: A diagram view, similar to a normal class diagram representation and a tree view (see Chapter 2.3.1, 2.3.2, and 2.3.5 in [31]). Both views are depicted in Figure 2.1 for a well-known metamodel example [37].

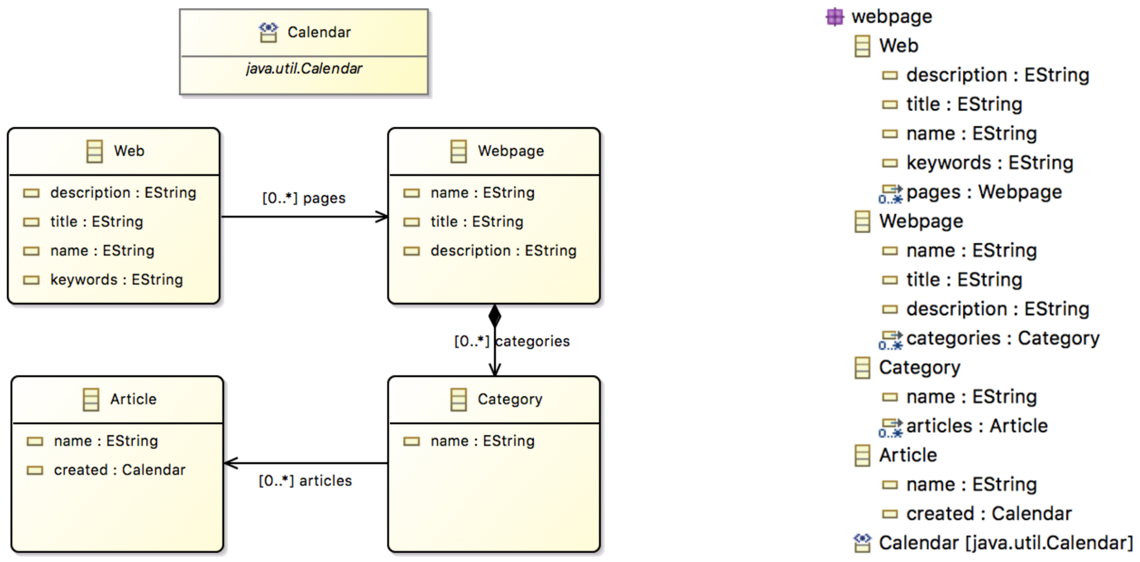


Figure 2.1.: Two views on an identical Ecore metamodel: The diagram view on the left and the tree view on the right [37].

2.6.1. Ecore Meta-Metamodel

The Ecore meta-metamodel is explained best with the Ecore kernel (see Figure 2.2). The Ecore kernel is a simplified subset of the Ecore meta-metamodel. The Ecore kernel defines four types. *EClass*, *EAttribute*, *EReference*, and *EDataTypes*. The type *EClass* models classes as known from the Java language. An *EClass* has a name as well as any number of supertypes, *EAttributes*, and *EReferences*. Similar to an UML class diagram *EAttributes* and *EReferences* can be used to model fields. Both types *EAttribute* and *EReference* have a name. However, while the referenced type of an *EReference* is an *EClass*, the *EAttribute* references an *EDataType*. *EDataTypes* model simple types that are not fully modeled as *EClasses*. *EDataTypes* only have a name (see Chapter 2.3.1 and 5.2 in [31]).

In the full Ecore meta-metamodel, *EAttribute* and *EReference* both extend the type *EStructuralFeature*, while *EClass* and *EDataType* extend the type *EClassifier*. Both *EStructuralFeature* and *EClassifier* are *ENamedElements*. *ENamedElements* and all other elements of the Ecore meta-metamodel are *EObjects* (see Chapter 5.3 and 5.5 in [31]). The full Ecore hierarchy can be seen in Figure A.2.

The Ecore hierarchy depicts many other elements of the Ecore meta-metamodel, for example, *EPackages*. *EPackages* contain *EClassifiers* and other *EPackages*. Every Ecore metamodel contains a specific *EPackage* as the root element. Therefore, *EPackages* define the basic tree structure of Ecore metamodels (see Figure 2.3). While the structural features of *EClasses* are modeled as *EStructuralFeatures*, behavioral features are modeled as *EOperations*. *EOperations* themselves only represent the signatures of the behavioral features. It is

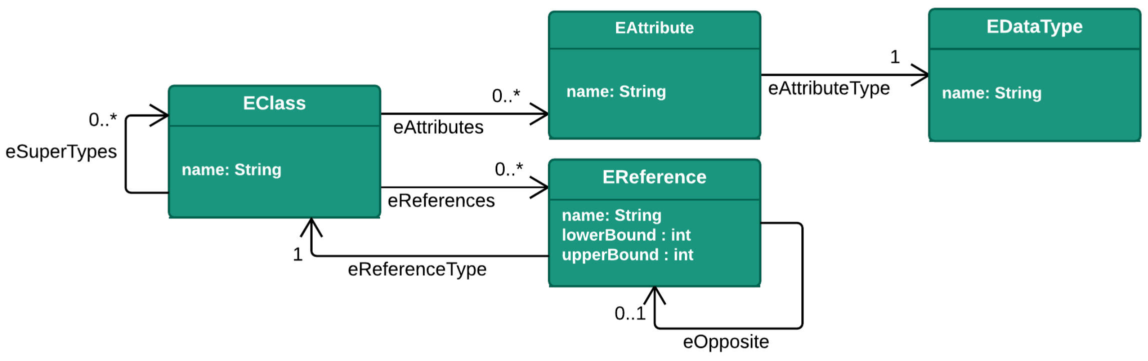


Figure 2.2.: UML Class Diagram of the Ecore kernel, a simplified subset of the Ecore meta-metamodel (see Chapter 5.2 in [31]).

possible to annotate an *EOperation* with its implementation details through *EAnnotations*. *EOperations* are contained by *EClasses*. As *EDataTypes*, *EOperations* are *ETypedElements* (see Figure A.2) (see Chapter 5.4 and 5.6 in [31]).

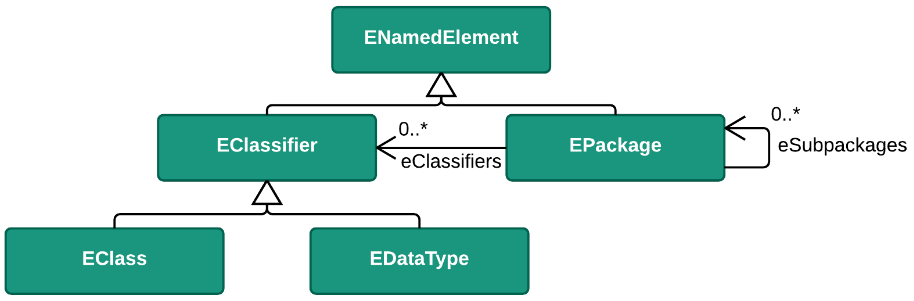


Figure 2.3.: UML Class Diagram of the *EPackage* Structure, which defines the basic tree structure of Ecore metamodels (see Chapter 5.6 in [31]).

2.6.2. Ecore Model Code

EMF defines a mapping from Ecore elements to Java interfaces. This mapping gives the possibility to generate Java code from an Ecore metamodel. Java code generated from an Ecore model is called model code. To generate model code, one has to create a *Generator Model*. The *Generator Model* is an Ecore model that is linked to an Ecore metamodel and contains additional information for the code generation of the metamodel, such as, the output location of the generated code (see Chapter 2.4 in [31]).

EMF defines a set of generator patterns, which are used during the model code generation. Every *EClass* is represented in the model code through two Java types: An interface and an implementation class. When generating model code from an *EPackage*, the code is organized into three Java packages. The first Java package is named after the *EPackage* itself and contains the interfaces that represent the *EClasses*. The second package is a subpackage of the first package and is named `impl`. It contains the implementation classes of the interfaces. The third package is also a subpackage of the first package and is named `util`. It contains a switch class and an adapter factory. They can be used to attach adapters to the modeled classes (see Chapter 10 in [31]). Additionally to these types, the Eclipse Modeling Framework also generates two interfaces and their implementations for the package: The package and the factory. While the package provides constants and convenient methods for the metadata of the *EPackage*, the factory allows creating instances of the packages implementation classes (see Chapter 10.8 in [31]). *EOperations* are represented in the generated model code through method signatures in the interfaces of their containing *EClasses* and methods in the relating implementation classes. *EStructuralFeatures* are represented in the model code through fields in the implementation classes of their containing *EClasses*. Additionally, EMF generates access methods for those fields, which are declared in the interfaces and implemented in the implementation classes (see Chapter 10.1, 10.2, and 10.5 in [31]).

When generating model code from a simple *EPackage* named `main` containing one *EClass* named `Person`, which contains one *EAttribute* named `name`, the model code contains three Java packages: `main`, `main.impl`, and `main.util`. The *EClass* is represented through the two Java types: `Person` and `PersonImpl`. These types are depicted in Figure 2.4. The interface `Person` declares the signatures of the access methods `getName()` and `setName()`. The class *PersonImpl* implements those access methods and contains the field `name` and its default value in another field.

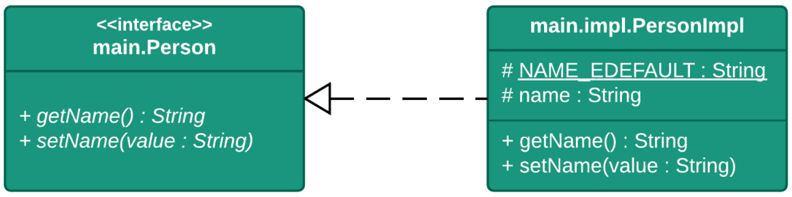


Figure 2.4.: Simplified UML Class Diagram of the model code which was generated from an *EPackage* named `main` containing one *EClass* named `Person`, which contains one *EAttribute* named `name`.

2.6.3. Ecore Functionality

As mentioned in the introduction (see Chapter 1), the model code generated from Ecore metamodels has several differences compared to arbitrary Java code. For example, it distinguishes between interface and implementation for every type. That means all methods are declared in an interface and implemented in a correlating class. While this is one feature that can be described as Ecore functionality, Ecore model code also contains additional features (see Chapter 2.4.1 in [31]).

Classes of Ecore model code extend the interface `Notifier`. This allows the support of model change notifications for every object of the model code. The model change notification system is similar to the observer design pattern. This system allows, for example, observing objects to update views or other objects (see Chapter 2.4.2 and 2.5.1

in [31]). Additionally, classes of the Ecore model code contain the methods `eContainer()` and `eResource()`, which allow access to the containing objects and resources of the class. This is a part of the EMF persistence API. It is a powerful mechanism for the management of object persistence which allows saving objects with an XMI serializer (see Chapter 2.4.2 and 2.5.2 in [31]). Another feature is the reflectivity API. Classes of the Ecore model code contain the methods `eGet`, `eSet`, `eIsSet`, `EUnset`. This allows manipulating objects through a model for generic access (see Chapter 2.4.2 and 2.5.3 in [31]). This thesis refers to all of these features as Ecore functionality.

3. Running Example

This chapter introduces a minimal Java code that is used throughout the thesis as an example explaining the integration of Ecore functionality into Java code. The simplicity of the code allows explaining complex processes on a very basic level and makes them easier to understand.

The code contains a package, three types, a specialization relation, and a realization relation. This is depicted in an UML class diagram in Figure 3.1. The package is called `company` while the types are called `Person`, `Customer`, and `BusinessPerson`. The class `Person` contains a private field of type `String`, which is called `name` and a public access method for that field called `getName()`. `BusinessPerson` is an interface which declares one public method called `generateReport`. The method takes one parameter of type `java.util.Date` which is called `day`. The class `Customer` extends the class `Person` and implements the interface `BusinessPerson`. It inherits the method `getName()` from `Person` and implements the method `generateReport()` from `BusinessPerson`. Additionally, the class `Customer` contains a private field of type `int`, called `customerID` and a public access method for that field called `getCustomerID()`.

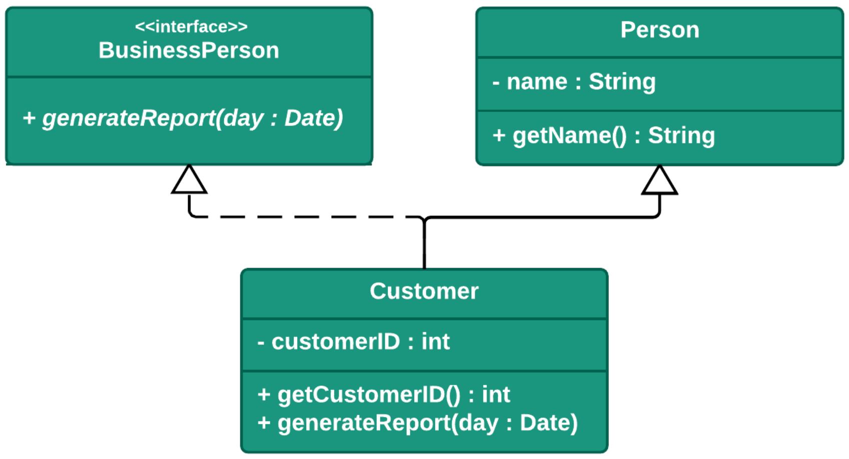
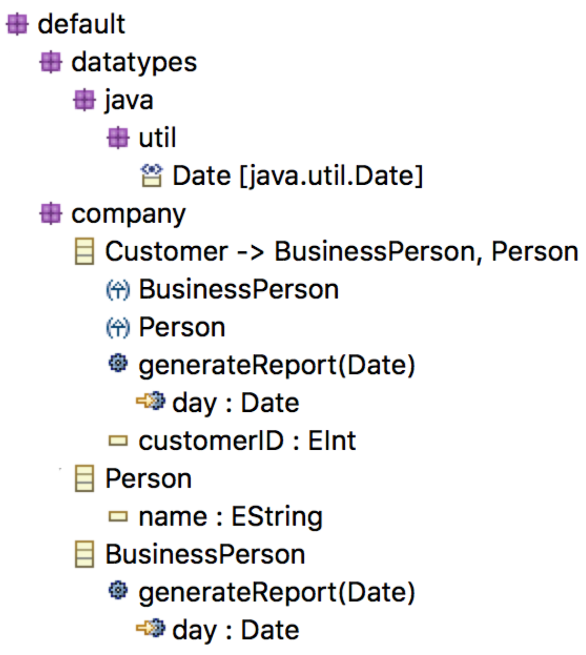


Figure 3.1.: The UML class diagram of the package `company`, containing the classes `Person` and `Customer` as well as the interface `BusinessPerson`.

3.1. Matching Ecore Metamodel

As previously explained, we developed an approach for the integration of Ecore functionality into ordinary Java code. A fundamental part of this approach is to find an Ecore representation of the Java code. Figure 3.2 shows the tree view of an Ecore metamodel. The root *EPackage* of the Ecore metamodel is called `default`. It contains two *EPackages*: `datatypes` and `company`. The *EPackage* `company` contains three *EClasses*: `Customer`, `Person`, and `BusinessPerson`. The *EClass* `Person` only contains an *EAttribute* of type *EString*, called `name`. The *EClass* `BusinessPerson` contains one *EOperation* called `generateReport`, which takes one *EParameter* of type `Date` called `day`. The *EClass* `Customer` contains two references to *EGeneric Super Types*. One references the *EClass* `Person`, the other references the *EClass* `BusinessPerson`. It also contains the same method as the *EClass* `BusinessPerson` and an additional *EAttribute* called `customerID`, which is of type *EInt*. The *EPackage* `datatypes` contains another *EPackage* called `java`, which contains the *EPackage* `util`. The *EPackage* `util` contains one *EDataType* called `Date`. This is the type referenced by the *EParameters* of both methods in `Customer` and `BusinessPerson`. It has the instance type name `java.util.Date`.



```
default
  datatypes
    java
      util
        Date [java.util.Date]
  company
    Customer -> BusinessPerson, Person
      BusinessPerson
      Person
      generateReport(Date)
        day : Date
      customerID : EInt
    Person
      name : EString
    BusinessPerson
      generateReport(Date)
        day : Date
```

Figure 3.2.: The extracted Ecore metamodel of the running example containing the *EClasses* `Person`, `Customer`, and `BusinessPerson` as well as the *EDataType* `Date`.

If we now compare Figure 3.1 and Figure 3.2, we can see many similarities: The Java package `company` match the *EPackage company* of the Ecore metamodel, while the Java types match the *EClasses* of the Ecore metamodel. The methods called `generateReport()` with their parameters match the *EOperations* with their *EParameters*, and the fields match the *EAttributes* of the Ecore metamodel. While the Ecore metamodel does not differentiate between specialization and realization relations, its super relations match the super relations of the class diagram. However, there are some differences between Figure 3.1 and Figure 3.2. The Ecore metamodel contains the additional packages `default`, `datatypes`, `java`, and `util`. The *EDataType* `Date` is modeled as an element, while the class diagram does not contain an element for the type `java.util.Date`. The class diagram also contains access methods, while the Ecore metamodel does not contain corresponding access *EOperations*.

The comparison of Figure 3.1 and Figure 3.2 suggests that it is possible to extract an Ecore metamodel from Java code by representing the different elements of the implicit Java metamodel through model elements from the Ecore meta-metamodel. The model elements have to be arranged in a hierarchical structure that matches the structure of the implicit Java model.

3.2. Code with Integrated Ecore Functionality

Let us now look at a simplified example for the integration of Ecore functionality into Java code. Figure 3.3 depicts another UML class diagram. The class diagram contains the class `Person` from Figure 3.1. It also contains three other types: `PersonWrapper`, `EPerson`, and `EPersonImpl`. The interface `EPerson` and the class `EPersonImpl` were generated from the *EClass* `Person` from Figure 3.2. That means they are part of the model code of the Ecore metamodel from Figure 3.2. The class `PersonWrapper` implements the interface `EPerson`. It also has a reference to the class `EPersonImpl`. That means the class `PersonWrapper` implements all Ecore functionality of the interface `EPerson` and delegates the implementation of this functionality to the class `EPersonImpl`. The class `Person` inherits that functionality because it extends the class `PersonWrapper`.

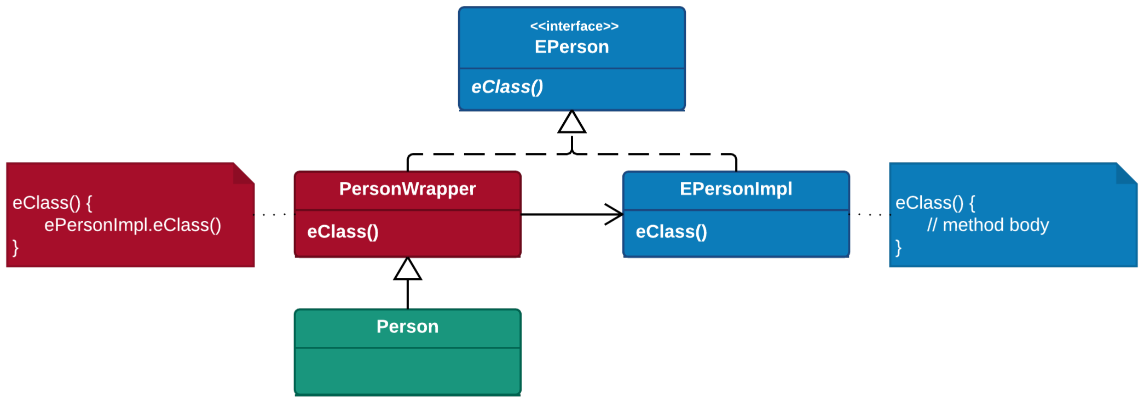


Figure 3.3.: UML class diagram that depicts an example of the integration of Ecore functionality into Java code.

That means figure 3.3 shows a simplified pattern to integrate Ecore functionality into Java code while using the model code of an Ecore metamodel, which closely represents the original Java code. This is the base concept for *Ecorification* process.

4. Ecore Metamodel Extraction

Because the Java code *Ecorification* relies on an Ecore metamodel that represents the Java code, there is a need for a reverse engineering process that can automatically generate such an Ecore metamodel. This section introduces a reverse-engineering approach for the extraction of Ecore metamodels from Java code. At the core of the approach is a mapping from elements of the implicit Java metamodel and elements of the Ecore meta-metamodel. This mapping defines how Ecore metamodel elements are extracted from Java code. We define the mapping for a higher metalevel to allow its application to any instances of the metalevel (see Section 2.2). For example, the mapping from the Java class to the Ecore metaclass *EClass* can be applied for any class. Table 4.1 gives a tabular view of the mapping between the Ecore meta-metamodel and the implicit Java metamodel.

Implicit Java Metamodel	Ecore Meta-Metamodel
Package	EPackage
Type	EClassifier
Class	EClass
Interface	EClass
Enum	EEnum
Enum Constant	EEnumLiteral
Field	EStructuralFeature
Method	EOperation
Method Parameter	EParameter
Method Return Type	*EClassifier/EGenericType*
Throws Clause	*EClassifier/EGenericType*
Generic Type Parameter	ETypeParameter
Generic Type Argument	EGenericType
Generic Type Bound	EGenericType
Super Type Reference	*EClass & EGenericType*

Table 4.1.: Tabular view of the mapping from elements of the implicit Java metamodel to elements of the Ecore meta-metamodel. A cursive Ecore element means the corresponding Java element is represented through a reference to the cursive Ecore element.

The Ecore meta-metamodel is a simplified subset of UML. Therefore, the Ecore meta-metamodel is very similar to an UML class diagram (see Chapter 2.3.1 and 2.6.1 in [31]). However, the Ecore meta-metamodel has also its specific characteristics (see Figure 2.1), which makes it in many ways fundamentally different. Because of this relation to UML, the

Ecore meta-metamodel, and the implicit Java metamodel are very similar, but not perfectly identical. That means there is no perfect mapping between the Ecore meta-metamodel elements and the implicit Java metamodel. As a result, not all features of Java code can be extracted, but the number of features that can be extracted allows the extraction of metamodels that accurately represent the Java code.

This chapter describes the extracted features of the Java language, their Ecore counterparts and how they are extracted. First, the extraction of the package structure is explained in Section 4.1. Second, the extraction of reference types is outlined in Section 4.2. Third, Section 4.3 discusses the extraction of primitive types. The extraction of methods and fields are explained in Section 4.4 and Section 4.5. Next, Section 4.6, outlines the extraction of inheritance and realization relations between classes. Section 4.7 explains the problem with custom exception and error classes. Section 4.8 outlines how modifiers are treated during the extraction. The sections 4.9 and 4.10 address the special cases where generic types or wildcard types are extracted. Next, Section 4.11, covers the selective extraction of an Ecore model from Java code. Finally, Section 4.12 summarizes the Ecore metamodel extraction.

4.1. Package Structure

The Java language organizes programs as sets of packages. Packages use a hierarchical package structure to organize related packages and types (see §7 in the JLS [12]). The Ecore meta-metamodel encapsulates types in *EPackages*, which are similar to the Java packages. *EPackages* and Java packages also serve a similar purpose conceptually.

In both Ecore and Java, the name of the package does not have to be unique. Java distinguishes packages through their fully qualified name, which is the hierarchical dot-separated concatenation of their names and the names of their super packages (see §6.7 in the JLS [12]). In an Ecore Metamodel, uniqueness is guaranteed through a unique Uniform Resource Identifier (URI). The URI usually also contains a concatenation of the hierarchical package names. Additionally, a namespace prefix is used to define the corresponding namespace prefix (see Chapter 5.6 in [31]). Because of these similarities, every Java package can be extracted directly from the project as an Ecore *EPackage*. The URI of an extracted *EPackage* consists of the project name, from which the package was extracted, and the fully qualified name of the package.

Ecore metamodels require one root *EPackage* which contains all other *EPackages*. This requirement does not exist in the Java language. A Java project can contain multiple source folders with multiple top level packages. Because of this difference, the Java default package is extracted as root package. As a result, the default *EPackage* is generated additionally to the extracted *EPackages*, when extracting a package structure from Java code to generate an Ecore metamodel. A second additional *EPackage* that is generated, is the data type *EPackage*. It is required to contain a hierarchy of *EDataTypes*. This behavior is explained in 4.2.1. For now, the only important thing is that this data type *EPackage* is additionally created to the other *EPackages*. The data type *EPackage* is a direct subpackage of the default *EPackage*.

That means every package in a target Java project is extracted as an *EPackage*. A class with the package structure `main.model`, `main.view.gui`, and `main.controller` will lead to the extraction of seven *EPackages*. Five of them are the counterparts to the Java packages: `main`, `model`, `view`, `controller`, and `gui`. The default package is extracted as the root package of the Ecore metamodel. The last *EPackage* is the *EPackage* `datatypes` for the data type hierarchy. This extracted *EPackage* structure can be seen in Figure 4.1.

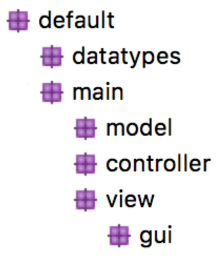


Figure 4.1.: The exemplary package structure of an extracted Ecore metamodel.

4.2. Reference Types

The Java language differentiates between two kinds of types: primitive types and reference types. Primitive types are either numeric types or boolean types. Reference types are either classes, interfaces, or array types (see §4, §4.1, §4.2, and §4.3 in the JLS [12]). This chapter covers only the extraction of reference types.

The Ecore meta-metamodel has an equivalent to types called *EClassifiers*. *EClassifiers* can be *EClasses* and *EDataTypes* (see Chapter 5.5 in [31]). An *EClass* can be seen as the equivalent of both the class and interface in Java. The two properties abstract and interface of an *EClass* enable the differentiation between classes, abstract classes, and interfaces (see Chapter 5.5.1 in [31]). Therefore a class or an interface can be extracted as *EClass* with the appropriate property configuration (see Table 4.2).

Java Type	*EClass* Property Abstract	*EClass* Property Interface
Class	false	false
Abstract Class	true	false
Interface	true	true

Table 4.2.: The *EClass* property configuration for the extraction of classes and interfaces.

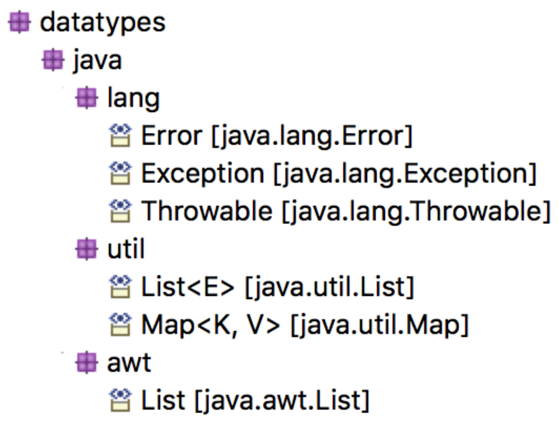
4.2.1. External Types

We define external types as types, which are referenced in a Java project or code but not declared in that project or code. Types are referenced by attributes, method parameters, method return types, throws declarations, inheritance relations, and realization relations. Examples of such external types are classes like `java.util.Date`. External types cannot be extracted as *EClasses* because they are not declared in the Java code that is used to extract a metamodel. Therefore, these external types have to be treated differently during the extraction.

As previously mentioned, *EClassifiers* can be divided into *EClasses* and *EDataTypes* (see Chapter 5.5 in [31]). *EDataTypes* represent a single piece of data, while *EClasses* offer more functionality than that. An *EDataType* in the Ecore meta-metamodel models one Java type. This Java type can either be a reference type or a primitive type (see Chapter 5.5.2 in [31]). *EDataTypes* are usually used to reference to Java types that are not modeled as *EClasses* in the Ecore meta-metamodel. Therefore the metamodel extraction uses *EDataTypes* for the extraction of external types.

Some Java types have correlating predefined *EDataTypes* (see Table A.1). These types are not added to the Ecore metamodel as model elements. They are simply referenced. These types can be identified by their fully qualified name (see §6.7 in the JLS [12]). All external types which do not have correlating predefined *EDataTypes* get extracted as custom *EDataType*, which is added to the metamodel and then referenced. The name of the custom *EDataType* is the name of the external type, while the instance type name of the *EDataType* is the full name of the external data type, including the package declaration. A reference to the external type `java.util.Date` is extracted as *EDataType* with the name "Date" and the instance type name "java.util.Date".

Custom *EDataTypes* have to be contained in *EPackages* of the metamodel. Because *EDataTypes* whose names are identical cannot be contained in the same *EPackage* (even if their instance type names are different), a package hierarchy has to be built for *EDataTypes* to avoid name collisions while extracting data types. This hierarchy is not part of the package hierarchy of the original Java code. An example of such package hierarchy can be seen in Figure 4.2.



```
datatypes
  java
    lang
      Error [java.lang.Error]
      Exception [java.lang.Exception]
      Throwable [java.lang.Throwable]
    util
      List<E> [java.util.List]
      Map<K, V> [java.util.Map]
    awt
      List [java.awt.List]
```

Figure 4.2.: Example for a data type hierarchy in an Ecore metamodel.

4.2.2. Enums

In the Java language, enums are a unique form of classes (see §8.9 in the JLS [12]). The Ecore counterpart for the enum in the Ecore meta-metamodel is the *EEnum*. The counterpart for the enum constants (see §8.9.1 in the JLS [12]) of an enum is the *EEnumLiteral*. While an enum in the Java language is seen as a class, the *EEnum* is not an *EClass* in the Ecore meta-metamodel. An *EEnum* is an *EDatatype* (see Chapter 5.5.2 in [31]). Because of this, the functionality of an *EEnum* is very limited compared to a Java enum. The *EEnum* is just a data type, while a Java enum can be treated in many ways like as class: It can have own methods, attributes, and constructors. Additionally, it can implement interfaces. All this functionality is not available for an *EEnum* [31].

Because of these differences, an enum can only be partially extracted when mapping it directly to an *EEnum*. That means an enum gets extracted as *EEnum*. The *EEnum* will contain *EEnumLiterals* equivalent to the enum constants of the enum. The *EEnum* will not contain the methods of the enum. To extract an enum with its methods, fields, constructor and interface relations, a single enum has to be extracted as one *EEnum* and one *EClass*. The *EEnum* will be extracted the same way it was before: With *EEnumLiterals* equivalent to the enum constants of the enum. The *EClass* contains the features of the enum that cannot be represented in an *EEnum* but can be represented in an *EClass*. It contains the Ecore counterparts of the methods, fields, and interface relations, but no *EEnumLiterals*. Because an Ecore metamodel cannot contain an *EClass* and an *EEnum* with the same name, a special naming scheme has to be employed to differentiate between the *EClass* and the *EEnum*, while distinctively associating the *EClass* to the *EEnum*. For example, an enum called `Color` could be extracted as *EEnum* `Color` and *EClass* `ColorEClass`. When choosing this extraction approach, it is important to avoid name collisions between the *EClass* and other *EClassifiers*. One way to avoid name collisions is to use an index, which increments as long as there is a name collision.

4.2.3. Nested Types

The Java language gives the possibility to nest class and interface types (classes, interfaces, and enumerations) into other class and interface types. A nested type is a type that is declared in the body of another type (see §8.5 in the JLS [12]). Functionality for nesting *EClassifiers* is not provided by Ecore meta-metamodel. As a result, inner types have no direct counterpart for the extraction.

Nested types can either be extracted as normal types with a special naming scheme, or they can be extracted in an additional subpackage named after their outer types (see Figure 4.3). A third choice would be to not extract nested types at all.

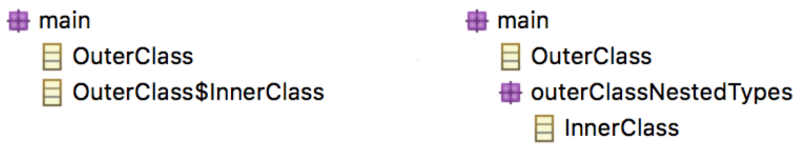


Figure 4.3.: Two extraction options for nested types. Special naming scheme on the left and additional subpackage on the right.

Extracting a nested type like a normal one with a special naming scheme (for example `OuterTypeInnerType` for the type `InnerType` nested in the type `OuterType`) can lead to name collisions if there already exists an *EClassifier* with the combined name. Extracting nested types in an additional subpackage named after their outer type (optionally with a special suffix) can lead to package name collisions if there already exists an *EPackage* with that name. Name collisions can be avoided through adding special suffixes to the names. For example, an index that increments as long as there still is a name collision. The main difficulty with this process is to retain the affiliation between the indexed *EClassifier* or *EPackage* and the outer type. That means it has to be possible to distinguish the nested, indexed *EClassifier* or *EPackage* from its similarly named counterpart. This thesis chooses to extract nested types in an additional subpackage.

4.2.4. Array Types

There is no direct support for arrays provided by the Ecore meta-metamodel. When manually building an Ecore metamodel, collections are modeled through multiplicities (e.g. `0..*`). This is realized in the generated code of a metamodel through *EList* instances [31]. That means there is no direct counterpart for Arrays when extracting Ecore metamodels from Java code.

If one wants to use an array in an Ecore metamodel, Eclipse Modeling Lead Ed Merks proposes to define an *EDataType* which has the Java syntax for the array as its instance type name. If one generates code from this metamodel, it is important to specialize the creation methods for the array data types in the implementation class of the package factory [25].

This approach is chosen for the extraction of array types. A one-dimensional character array will be extracted as *EDataType* with the name "charArray" and the instance type name "char[]". A three-dimensional String array will be extracted as *EDataType* with the name "StringArray3D" and the instance type name "java.lang.String[][][]". The instance type name of an *EClassifier* represents the parameterized Java type that this *EClassifier* represents. See chapter 4.2.1 for the use of *EDataTypes* in the extraction process.

4.3. Primitive Types

As previously mentioned, the Java language differentiates between two kinds of types: primitive types and reference types (see §4 and §4.1 in the JLS [12]). Primitive types are either numeric types or boolean types (see §4.2 in the JLS [12]). While reference types were covered in the previous section, Section 4.2, this section covers primitive types.

The Ecore meta-metamodel itself offers predefined *EDataTypes* for the primitive types of the Java language. These primitive types can be seen in Table 4.3. It also offers *EDataTypes* for the object variants of the primitive data types and other commonly used classes (see Chapter 5.8 in [31]). The full list of predefined *EDataTypes* can be seen along with their Java counterparts in Table A.1. During the extraction, primitive types and their object variants are identified and then represented through the predefined *EDataTypes*, instead of creating own *EDataTypes* (see Section 4.2.1). Primitive types are identified by their specific keywords (see §3.9 in the JLS [12]), while their object variants are identified by their fully qualified name (see §6.7 in the JLS [12]).

Java Type	Ecore Type
boolean	EBoolean
byte	EByte
char	EChar
double	EDouble
float	EFloat
int	EInt
long	ELong
short	EShort

Table 4.3.: Predefined *EDataTypes* for primitive types and their Java counterparts (see Chapter 5.8 in [31]).

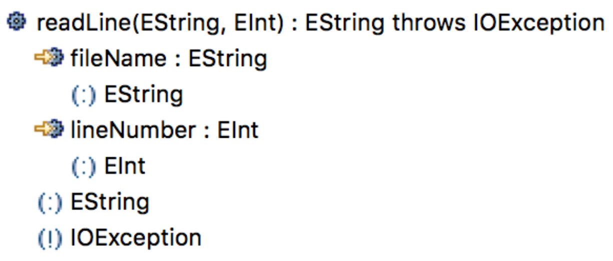
4.4. Methods

Reference types can declare methods in the Java language. Methods are declarations of executable code that can be invoked while passing a fixed number of parameters. Methods consist of a method header and a method body. The body contains the executable code, while the header declares the name of the method, the return type, the parameters and the throws clause (see §8.4 in the JLS [12]).

The Ecore meta-metamodel uses *EOperations* to model the behavioral features of *EClasses*. *EOperations* are contained in an *EClass*. *EOperations* contain *EParameters*, *EGeneric Exceptions*, and an *EGeneric Return Type*. All three reference *EClassifiers* (see Chapter 5.4 in [31]). *EOperations* are the Ecore counterpart to methods in the Java language. While it is possible to annotate an *EOperation* with implementation details, this thesis does not extract such details from methods. Methods are extracted as method signatures with their name, their return type, their parameters and their throws clauses (see §8.4 and §9.4 in the JLS [12]). They are represented as *EOperations* in the Ecore metamodel.

The return type of a method is a reference to a type (see §8.4.5 in the JLS [12]) and therefore are extracted as a reference to an *EClassifier*. Depending on whether this method return type references an external type or not, the *EOperation* return type references an *EClass* or *EDataType*. The method parameters are extracted as *EParameters*. Like the method parameters in the Java language (see §8.4.1 in the JLS [12]), *EParameters* have a name and a type, which they reference. The type of the *EParameter* references an *EClassifier*, which is depending on whether the parameter type references an external type or not, an *EClass* or *EDataType*. The throws clauses of methods reference exception types (see §8.4.6 in the JLS [12]). They are extracted as references to *EClassifiers*, which are, again, either *EClasses* or *EDataTypes*.

Figure 4.4 visualizes an *EOperation* that was extracted from a Java method with the signature `String readLine(String fileName, int lineNumber) throws IOException`. The *EOperation* contains an *EGeneric Return Type* which references the *EDataType EString*, two *EParameters* which reference the *EDataTypes EString* and *EInt* and an *EGeneric Exception* which references the *EDataType IOException*.



```
readLine(EString, EInt) : EString throws IOException
  fileName : EString
    (:) EString
  lineNumber : EInt
    (:) EInt
  (:) EString
  (!) IOException
```

Figure 4.4.: Extraction example for a Java method called `readLine()`, which has two parameters, a return type and a throws clause.

4.4.1. Access Methods

Fields are commonly accessed through access methods. Access methods are divided into accessors, which are used to get the values of fields and mutators, which are used to set the values of fields. Accessors are commonly referred to as getters, while mutators are commonly referred to as setters [14]. When generating code from an Ecore metamodel, EMF automatically generates access methods for the fields generated from the *EStructuralFeatures* (see Chapter 10.1.2 in [31]). For that reason, access methods should not be extracted as *EOperations* when extracting the correlating fields, because that would lead to duplicate access methods when generating code from the extracted Ecore metamodel.

The Java language does not differentiate between access methods and ordinary methods, therefore, access methods have to be detected from their method signature. This can be achieved by checking whether a method signature matches the design patterns for fields in the Java Beans Specifications (See Chapter 8.3 in [14]). While access methods for simple fields and boolean fields are generated automatically by EMF, access methods for indexed fields are not. That means access methods indexed fields can be extracted as *EOperations* because they are not automatically created when generating Java code from an Ecore metamodel.

Not extracting access methods can lead to problems when they have more than just the functionality that access methods usually contain. In this case, the access method generated by EMF from the extracted Ecore metamodel will be different from the original access method, because the generated access methods contain no additional functionality. An example for such additional functionality can be as simple as rounding the return value in an accessor or validation code in a mutator.

4.4.2. Constructors

While constructors play an essential role in the object-oriented Java language (see §8.8 in the JLS [12]), there is no counterpart in the Ecore meta-metamodel. Initial values of *EStructuralFeatures* can be declared with default value properties (see Chapter 5.3 in [31]). Moreover, because the Ecore metamodel is not modeling implementation details, there is no other use case for constructors. When generating code from an Ecore metamodel, the generated classes are instantiated with factory classes that only call the parameterless constructors (see Chapter 10.8 in [31]). The fields of those generated classes get their initial values with static default value fields (see Chapter 10.2 and 10.3 in [31]). They also can be set after the object creation with access methods. As a result, there is no way to represent constructors in the Ecore meta-metamodel.

Representing constructors in an Ecore metamodel is not possible, but constructors and especially parameterized constructors are very important for the instantiation of classes. This difference between the Ecore meta-metamodel and the implicit Java metamodel will lead to problems during the *Ecorification* process.

4.5. Fields

Fields are members of classes and interfaces in the Java language. They declare the variables of their types. Fields have an identifier, also called name, to distinguish between the fields of a type and a reference to a type (see §8.3 and §9.3 in the JLS [12]).

The Ecore counterpart for fields in the Java language are *EStructuralFeatures*. They are divided into *EAttributes* and *EReferences* (see Chapter 5.3 in [31]). This concept is similar to UML class diagrams, where fields can either be attributes or references. However, *EAttributes* and *EReferences* are not arbitrary interchangeable. *EReferences* are used for *EStructuralFeatures* that reference *EClasses* (see Chapter 5.3.2 in [31]), while *EAttributes* are used for *EStructuralFeatures* that reference *EDataTypes* (see Chapter 5.3.1 in [31]). This behavior is depicted in Figure 4.5.

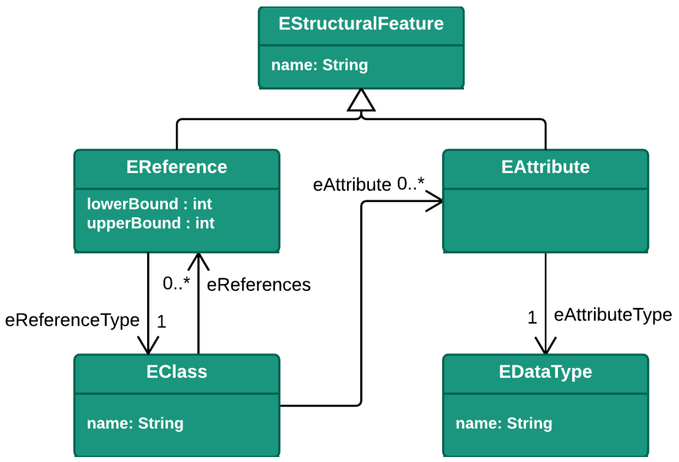


Figure 4.5.: Simplified class diagram of the *EStructuralFeature* architecture (see Chapter 5.3 in [31]).

A Java field gets extracted as *EReference* when the field references a type that is part of the Java code that is used to extract the Ecore metamodel. That means there is an *EClass* representing that type. A Java field gets extracted as *EAttribute* when the field references an external type (see Section 4.2.1), which is not part of the Java code (for example classes like `java.util.Date`). Figure 4.6 shows an example for the extraction of two fields. While the *EStructuralFeature* `date` references the *EDataType* `Date`, the *EStructuralFeature* `organizer` references the *EClass* `Organizer`. Therefore `date` is an *EAttribute* and `organizer` is an *EReference*.

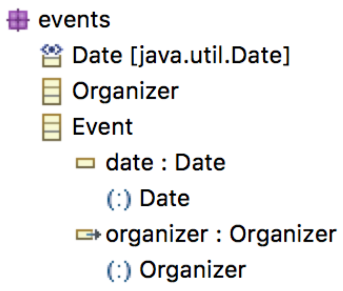

```
events
  Date [java.util.Date]
  Organizer
  Event
    date : Date
      (:) Date
    organizer : Organizer
      (:) Organizer
```

Figure 4.6.: Extraction example for fields.

Because fields store the internal state of a type, it is not always useful to extract fields. An alternative option is to extract the access methods of fields, but not the fields themselves. As mentioned as in Section 4.4.1, EMF automatically generates access methods for the fields generated from the *EStructuralFeatures* (see Chapter 10.1.2 in [31]). That means fields should not be extracted, when their access methods are extracted and access methods should not be extracted when their fields are extracted. For the integration of Ecore functionality into Java code, it is better to extract the fields while not extracting the access methods because the fields in model code generated from an Ecore metamodel have additional functionality compared to arbitrary fields.

4.6. Specialization and Realization Relations

The Java language uses the two powerful concepts specialization and realization. Specialization relations allow extending another class, which is called superclass (see §8.1.4 in the JLS [12]). This concept is also referred to as inheritance. Realization relations allow implementing an interface, which is called super interface (see §8.1.5 and §9.1.3 in the JLS [12]). Specialization and realization are often referred to as super-relations.

The Ecore metamodel has a similar concept. As previously explained, the Ecore meta-metamodel uses *EClasses* to represent both classes and interfaces from the Java language. An *EClass* can have multiple supertypes with the *eSuperTypes* reference. While there is the possibility to extend *EClasses* and implement *EClasses* that are interfaces, there is no distinguishing between the specialization and realization relation. There is just the supertype relation through the *eSuperTypes* reference (see Chapter 5.2 and 5.5.1 in [31]).

The Ecore meta-metamodel supports multiple inheritance. That means it is possible for one *EClass* to extend (and implement) multiple *EClasses*. While the Java language allows a type to implement multiple interfaces, inheritance is limited to one superclass per class. Therefore, multiple inheritance is not needed to represent Java code in an Ecore metamodel. As described in Section 4.2.2, *EEnums* are not a special form of *EClasses*. *EEnums* are *EDataTypes*. Because *EDataTypes* do not allow specialization or realization relations, *EEnums* cannot inherit from *EClasses* or implement *EClasses*. This behavior is a fundamental difference to the Java language, where Enums are a special form of classes and have the ability to both specialization and realization relations.

The differences between the use of super-relations in the Java language compared to the Ecore meta-metamodel are listed in Table 4.4. The table shows whether a type or *EClassifier* can have such a relation to a super type.

Relation	Class	Interface	*EClass*	Enum	*EEnum*
Specialization	yes	no	yes	yes	no
Realization	yes	yes	yes	yes	no

Table 4.4.: Specialization and realization relations for the different types of the Java language and the different *EClassifiers* of the Ecore meta-metamodel.

When extracting super-relations from Java code, classes and interfaces do not lead to problems. Specialization relations are extracted as super-relation to an *EClass* and realization relations are extracted as super-relation to an *EClass* which is an interface (see Section 4.2). The extraction of super-relations of an enum is not possible. Enums are extracted as *EEnums* which do not have that functionality. As a result, these relations cannot be represented in an Ecore metamodel, as long as the solution of extracting an enum as both an *EEnum* and an *EClass* is not implemented (see Section 4.2.2).

In an Ecore metamodel, the supertypes of an *EClass* have to be *EClasses* within the metamodel. As explained in Section 4.2.1, external types are not depicted as *EClasses*. That means the use of so-called external supertypes is not possible. This functionality is not a problem in the Java language. For example, it is possible to create a class that inherits from the project external class `java.util.LinkedList` of the Java API. This difference between the Ecore meta-metamodel and the Java language is the reason why super-relations to external supertypes cannot be extracted. These relations are lost during the extraction and cannot be represented in an Ecore metamodel.

4.7. Custom Exception and Error Classes

In the Java language the class `java.lang.Throwable` is the superclass of all errors and exceptions. Extending `java.lang.Throwable` or a subclass of it allows creating custom exception and error classes. These custom exception and error classes can be used in several contexts. One of which is its use in a throws clause of a method (see Section 4.4).

When extracting custom exception and error classes, the specialization relation that makes them a subclass of `java.lang.Throwable`, cannot be extracted in most cases because of the problem with external super types (see Section 4.6). The only case when this specialization relation can be extracted is when extracting an Ecore metamodel from the Java API package `java.lang`. In this case the class `java.lang.Throwable` would not be an external type, which would allow extracting the specialization relation. When the specialization relations of custom exception and error classes are not extracted, but the exception and error classes themselves are extracted, the extracted metamodel generates invalid Java code whenever custom exception and error classes are used in a throws clause. The reason for this is that the custom exception and error classes translate to Java classes that do not extend `java.lang.Throwable`. This results in compiler errors in every method where they are used in a throws clause because the language only allows referencing subtypes of `java.lang.Throwable` in throws clauses.

Because of this reason, custom exception and errors classes should not be extracted, or at least only on demand. When they are not extracted as *EClasses*, they are generated as *EDataTypes* when they are referenced (see Section 4.2.1). That means when custom exception and errors classes are not extracted, the problem with external super types is avoided.

4.8. Modifiers

The Java language uses five types of modifiers: Class modifiers, field modifiers, method modifiers, constructor modifiers and interface modifiers. While some of them use the same modifier keywords, one modifier keyword can have multiple meanings depending on which type of modifier it is (see §8.1.1, §8.3.1, §8.4.3, §8.8.3, and §9.1.1 in the JLS [12]). A full list of the Java modifiers and their affiliation with the modifier types can be seen in Table 4.5.

Most of the Java modifier keywords cannot be represented in the Ecore meta-metamodel, simply because it has no similar counterpart for the specific Java modifier. Those modifier keywords, which can be represented, can often only be accordingly represented in a special context. For example, the keyword `final` can only be extracted as a field modifier.

This is not a problem for many of the modifier keywords because the Ecore meta-metamodel is not meant to model implementation details. An Exception is the *EAnnotation* which can be used to attach source code to an *EOperation*, which is later used during the model code generation from this Ecore metamodel (see Chapter 5.7.1 in [31]). Therefore all modifier keywords that deal with implementation details are not supposed to be extracted. We only extract modifier keywords that are relevant for the Ecore metamodel.

Keyword	Class	Field	Method	Constructor	Interface
public	yes	yes	yes	yes	yes
protected	yes	yes	yes	yes	yes
private	yes	yes	yes	yes	yes
abstract	yes	no	yes	no	yes
static	yes	yes	yes	no	yes
final	yes	yes	yes	no	no
synchronized	no	no	yes	no	no
volatile	no	yes	no	no	no
transient	no	yes	no	no	no
native	no	no	yes	no	no
strictfp	yes	no	yes	no	yes

Table 4.5.: Java Modifier Keywords and to which modifier type they belong.

4.8.1. Access Level Modifiers

Access level modifiers are modifiers that manage access control. They influence the visibility of the type or member they are assigned to. This prevents from depending on unnecessary implementation details. The Java access level modifier keywords are `public`, `private` and `protected`. As seen in Table 4.5, they are available for every modifier type (see §6.6 in the JLS [12]).

There are no actual access level modifiers in the Ecore meta-metamodel. It also does not have any similar counterpart, but the Ecore meta-metamodel treats *EStructuralFeatures* as Java would treat private fields and *EOperations* as Java would treat public methods (see Chapter 5.3 and 5.3 in [31]). This behavior also translates into the model code: When generating code from Ecore metamodels, *EOperations* are created as public methods, while *EStructuralFeatures* are generated as private fields. For these private fields, accessor methods are generated (see Chapter 10.2, 10.3, and 10.5 in [31]).

As a consequence, every extracted field is represented as private *EStructuralFeature*, and every extracted method is represented as public *EOperation*. This leaves two options for the extraction: Either the extraction of all methods as *EOperations* and all fields as *EStructuralFeature*, even if their access level modifiers will not match the behavior of their generated counterparts, or restricting the extraction of elements depending on their access level modifiers. The second option would mean that only public methods and private fields are extracted.

An argument for the second option is the concept behind the Ecore metamodel. As previously explained, the Ecore meta-metamodel does model implementations details. That means private methods should not be extracted as *EOperations* because *EOperations* are not meant to model internal methods. The Ecore meta-metamodel does model some internal state through *EStructuralFeatures*. However, *EStructuralFeatures* are not meant to be accessed externally. Therefore public fields should not be extracted as *EStructuralFeatures* because *EStructuralFeatures* do not model exposed fields.

4.8.2. Modifier Keyword Abstract

The modifier keyword `abstract` offers the possibility to declare abstract classes, interfaces, and methods. Abstract methods provide a normal method signature, a return type and throws clause but do not have a method body. That means they do not provide an implementation. Abstract classes are classes that contain one or more abstract methods. Every subclass of an abstract class that is not abstract has to provide implementations for all abstract methods. All interfaces are implicitly abstract without an explicit declaration with the keyword `abstract` (see §8.1.1, §8.4.3.1, and §9.1.1.1 in the JLS [12]).

While the Ecore meta-metamodel certainly does have an abstract concept for *EClasses*, there are no abstract *EOperations* in the Ecore meta-metamodel as a counterpart to abstract methods in the Java language [31]. This means abstract methods can only be extracted as normal *EOperations* or cannot be extracted at all. When an abstract method is not extracted, and the extracted metamodel is used to generate model code, there is neither a method signature declaration in the interface nor a method in the implementation class of the model code. That means while the implementing methods of the abstract methods exist in the generated subclasses, the class itself does not know of their existence. When extracting an abstract method as a normal *EOperation*, the *EOperation* models a method declaration instead of a method signature declaration. That means when generating model code from the extracted metamodel, a method signature is declared in the interface, and this method is then implemented in the implementation class. While this solution allows calling the method which originally was abstract, this will lead to an `UnsupportedOperationException`, as long as the method body of the method in the implementation class is not modified. While this behavior is not identical to the behavior of an abstract method, it is the closest one can get to an abstract method in the model code. Additionally, the methods could be modified to throw a custom exception which clarifies that this method is representing an abstract method and should not be called. The custom exception would help to differentiate the behavior of these methods from other unimplemented methods.

For the Ecore representation of the modifier keyword `abstract` in the context of types see Section 4.2.

4.8.3. Modifier Keyword Static

The Java language differentiates between static elements and non-static elements using the modifier keyword `static`. That means one can create static fields and methods, which do not need object instantiation. Those attributes and methods can be accessed through the class directly (see §8.3.1.1 and §8.4.3.2 in the JLS [12]). The Ecore meta-metamodel has not such a static concept. As a result, attributes and methods can only be extracted as normal (object-oriented) *EStructuralFeatures* and *EOperations* or cannot be extracted at all.

4.8.4. Modifier Keyword Final

The keyword `final` has, like many other modifier keywords, a different effect depending on the context of its use. When used as a field modifier, the keyword `final` has the consequence that the field can only be initialized once. Also, it has to be initialized before or during the initialization of its declaring type (see §8.3.1.2 in the JLS [12]).

EStructuralFeatures in the Ecore meta-metamodel have a property called *changeable*. This property determines whether it is possible to set the *EStructuralFeature* externally. When generating model code from Ecore metamodels, unchangeable *EStructuralFeatures* will not have a mutator method. That means they can only be initialized internally (see Chapter 5.3, 10.2.7, and 10.3.7 in [31]). While this behavior is not identical to the Java modifier keyword `final`, it is similar enough to argue that `final` fields could be extracted as unchangeable *EStructuralFeatures* to keep the behavior of the extracted metamodel as close as possible to the original behavior of the Java code.

4.8.5. Modifier Keyword Transient

EStructuralFeatures contain the property *transient*. If this property is set, the *EStructuralFeature* is not part of the persistent state of an EObject. That means it is omitted from the serialization of the EObject (see Chapter 10.3 in [31]). This property is the counterpart to the `transient` keyword of the Java language. The Java `transient` modifier has the same function for fields: Preventing them from being serialized (see §8.3.1.3 in the JLS [12]). This means `transient` fields can be extracted as *transient EStructuralFeatures*.

4.9. Generics

Generics are a powerful feature of the Java language that allows programmers to create own generic type declarations. Classes, interfaces, and methods are considered generic if they use at least one type variable (see §8.1.2, §9.1.2, and §8.4.4 in the JLS [12]). A type variable is an unqualified identifier which is introduced by the declaration of a so-called type parameter (see §4.4 in the JLS [12]). Type variables can be used as normal types, for example as a field, type of a method parameter or method return type. They are limited to the scope in which they were declared.

The type parameter of a class, interface or method is a generic type that has to be specified when instantiating or inheriting from a generic class, calling a generic method or implementing a generic interface. This happens for all type parameters of a generic type or method through a list of generic type arguments. The generic type arguments denote a particular parameterization of the generic type (see §4.5 in the JLS [12]). That means they specify which types are used in the generic type or method. Generic arguments also can be type variables. Every type parameter has a so-called bound. The bound restricts the unknown type of the type parameter. The bound is either another type parameter or up to one class type with any number of interface types. A type parameter without an explicitly declared bound has the default bound `java.lang.Object` (see §4.4 in the JLS [12]).

The Ecore meta-metamodel was adapted to support generics when that feature was added to the Java language. The Ecore counterpart of a type parameter is an instance of the class *ETypeParameter*, which can be contained by an *EClass* or *EOperation*. A type parameter bound can be represented through any number of *EGeneric Bound Types*, which are *EGenericTypes*. That means an *EGeneric Bound Type* in the Ecore metamodel is either another *ETypeParameter* or any number of *EClassifiers* (see Figure 4.7). When an *ETypeParameter* has no *EGeneric Bound Type*, it is bound to *EObject*. The *ETypeParameters* of an *EClassifer* have to be specified by *EGenericArguments* whenever the *EClassifer* is referenced.

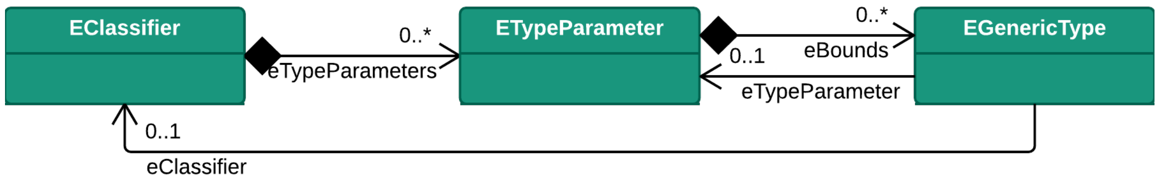


Figure 4.7.: UML class diagram that depicts how *ETypeParameters* are declared (see Chapter 21.1.2 in [31]).

EStructuralFeatures, *EOperations*, and *EParameters* inherit from the class *ETypedElement* the ability to reference an *EGenericType*. That means they have two type references: *eType* and *eGenericType*. The *eGenericType* reference is used instead of the *eType* whenever an *ETypeParameter* should be referenced instead of an *EClassifier*. The *eGenericType* reference references an *EGenericType* which references an *ETypeParameter* with the reference *eTypeParameter* (see Figure 4.8).

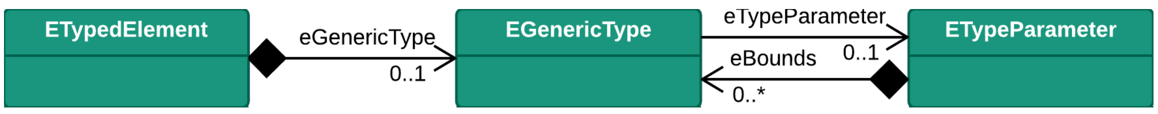


Figure 4.8.: A class diagram that depicts how an *ETypeParameter* is referenced from an *ETypedElement* (see Chapter 21.1.2 in [31]).

When referencing an *EClassifier* that has *ETypeParameters*, the *ETypeParameters* have to be specified by *EGenericArguments*. *EGenericArguments* are *EGenericTypes* that specify the *ETypeParameters* and are contained in a mutual *EGenericType*. As discussed previously, *ETypedElements* have a reference to an *EGenericType*. For other model elements, other references are available. For example, an *EOperation* uses its reference *eGenericExceptions* for its reference *eExceptions*, while an *EClass* uses its reference *eGenericSuperTypes* for its reference *eSuperTypes*. The *EGenericType* is then used to reference to any number of other *EGenericTypes*, which serve as *EGeneric Type Arguments* (see Chapter 21.1.2 in [31]). This behavior is depicted in Figure 4.9.

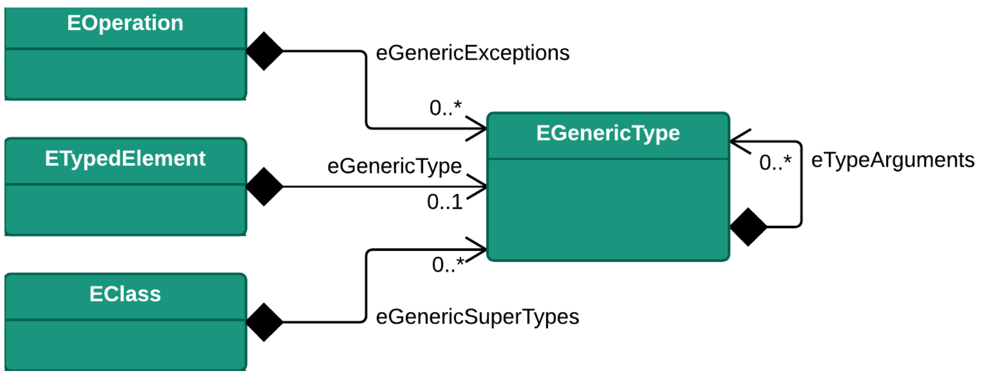
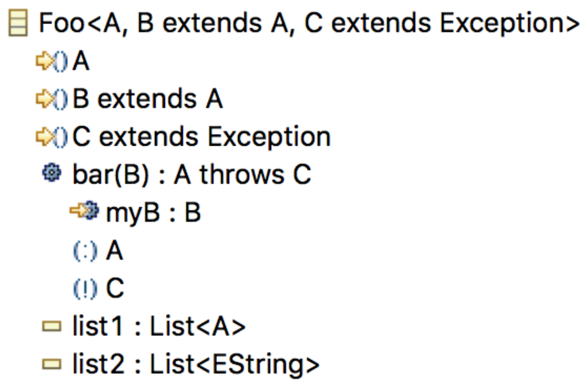


Figure 4.9.: A class diagram that depicts how *EGenericArguments* are specified (see Chapter 21.1.2 in [31]).

Because of the similarities of generics in the Ecore meta-metamodel and the Java language, the extraction of Ecore metamodels from generic Java code is conceptually very simple: A type parameter is extracted as an *ETypeParameter*, while a type parameter bound is extracted as the reference *eBounds*. The referenced types of a bound are extracted as *EGenericTypes*, which are referencing either an *EClassifier* or an *ETypeParameter*. Generic arguments of a reference to a type are extracted as *EGenericTypes*. They are contained in the *EGenericType* of the reference owners Ecore representation. For example, if a Java class contains an attribute which is parameterized with generic arguments, the *ETypeParameters* will be contained in the *EGenericType* of the *EClass* that was extracted from the Java class.

Figure 4.10 shows an example for the extraction of generics. The *EClass* `Foo` is extracted from a Java class that declares the three type parameters `A`, `B extends A`, and `C extends Exception`. The class contains two attributes of type `List<A>` and `List<String>` and a method with the signature `public A bar(B myB) throws C`.



```
Foo<A, B extends A, C extends Exception>
  A
  B extends A
  C extends Exception
  bar(B) : A throws C
    myB : B
    A
    C
  list1 : List<A>
  list2 : List<EString>
```

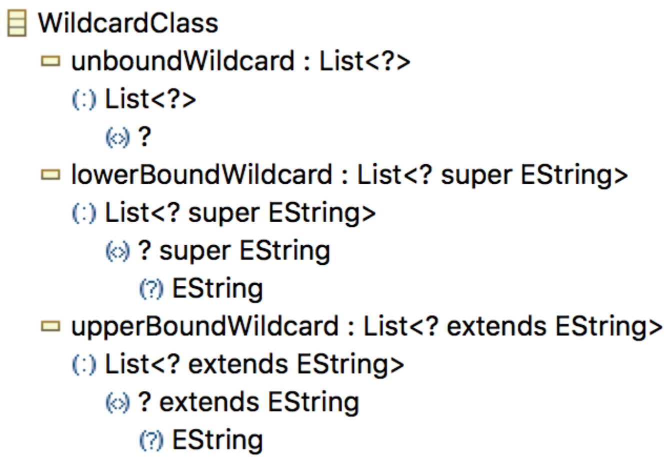
Figure 4.10.: An example metamodel that depicts how generic classes are extracted. The *EClass* `Foo` contains three *ETypeParameters*, two parameterized *EAttributes*, and a parameterized *EOperation*.

4.10. Wildcard Types

The Java language contains a particular type parameter called wildcard. It is represented by the question mark character and represents an unknown type. It can be used as a parameter type, field or local variable. It is not used as a type argument for generic methods, generic classes, or supertypes. A wildcard bound constrains the inheritance of the unknown type. A wildcard is either unbound, has an upper bound, or a lower bound. If a wildcard type has an upper bound, the type has to be a subtype of the type referenced by the upper bound. If a wildcard type has a lower bound, the type has to be a supertype of the type referenced by the lower bound. Similarly to normal type parameters, an unbound wildcard has the upper bound `java.lang.Object` (see §4.5.1 in the JLS [12]).

The Ecore metamodel is capable of representing the use of wildcard types. Wildcard types are represented, just as *EGenericArguments*, as *EGenericTypes*. While normal *EGenericArguments* either reference an *EClassifier* through the *eClassifier* reference or reference an *ETypeParameter* through the *eTypeParameter*, wildcard types do not use any of these references. An *EGenericType* without further references is an unbound generic type. Bound wildcard types are declared through referencing a bound type through either the reference *eUpperBound* or the reference *eLowerBound*.

Wildcard types are extracted as *EGenericTypes*. Wildcard bounds are extracted into the references *eUpperBound* and *eLowerBound*. The types referenced by the wildcard bounds are extracted as *EGenericTypes*, which are referencing either an *EClassifier* or an *ETypeParameter*. This extraction is depicted in Figure 4.11. It shows how three fields are extracted, which reference the type `java.util.List` using wildcard types. The first wildcard type is unbound, the second has a lower bound, and the third has an upper bound.



WildcardClass
unboundWildcard : List<?>
List<?>
?
lowerBoundWildcard : List<? super EString>
List<? super EString>
? super EString
EString
upperBoundWildcard : List<? extends EString>
List<? extends EString>
? extends EString
EString

Figure 4.11.: Example metamodel that depicts how wildcard types are extracted. The *EClass* contains three *EAttributes* which reference a generic *EDataType* using wildcard types.

4.11. Selective Extraction

When extracting Ecore metamodels from larger Java projects, it is not always useful to extract the whole project as one metamodel. If one wants to extract a metamodel from a submodule of a project, it is crucial to support selective extraction. Selective extraction means extracting an Ecore metamodel from specific packages or types and, moreover, selecting specific members of types.

Selective extraction leads to one problem: When a type references another type, which is not selected for the extraction, there is no representation to reference to for the Ecore representation of the first type. Consider the following example: A class `Alpha` has a field of type `Beta`. When selecting both `Alpha` and `Beta`, both get extracted as an *EClass* while the field gets extracted as *EReference* which references the *EClass* `Beta`. When selecting only the class `Alpha`, there is no *EClass* `Beta`. That means the *EReference* has no *EClass* to reference.

To avoid this, one has to identify all critical types. Critical types are all types that are not selected but referenced by another type which is selected. These critical types are then modeled as *EDataTypes*, which can be referenced by other *EClassifiers*. This means they are treated like external types (see Section 4.2.1). Because they are now treated like external types, specialization and realization relations (see Section 4.6) to the *EDataType* representation of a critical type are not possible. These external super-relations cannot be represented in an Ecore metamodel.

4.12. Summary

The extraction approach proposed by this thesis is based on a mapping between the Ecore meta-metamodel and the implicit Java metamodel. This mapping is depicted in Table 4.1.

Reference types like classes, abstract classes and interfaces are extracted without problems. They are modeled as *EClasses*. Exceptions are custom exception and error classes. They should not be extracted because their super relation to `java.lang.Throwable` cannot be extracted. Extracting them can cause errors during the model code generation because methods can only reference subtypes of `java.lang.Throwable` in their throws clauses. Enums are extracted, but extracting all of their functionality is complicated because an enum has to be modeled as one *EEnum* and one *EClass* to achieve that. One *EEnum* alone can only contain enum constants, which are modeled as *EEnumLiterals*. Nested reference types cause problems during the extraction. They are modeled as normal types either in a special *EPackage* or with a special naming scheme. Although array types have no direct counterpart in the Ecore metamodel, they are extracted as *EDataTypes*. Primitive types and their object variants are extracted without problems because EMF offers predefined *EDataTypes* that can be used to model them.

Methods and fields are extracted as *EOperations* and *EStructuralFeatures*. It is important to only extract the fields or their access methods, not both. That would lead to problems during the model code generation due to the duplicate access method declarations. Constructors are not extracted because there is no fitting counterpart in the Ecore meta-metamodel. While they are not needed in Ecore metamodels, this is a problem for the *Ecorification*. Specialization and realization relations are extracted as long as the referenced supertypes are not external types. When extracting modifier keywords, it depends on the keyword and the context whether they are extracted or not. The modifier keyword `transient` is extracted without problems as the *EStructuralFeature* property *transient*. The modifier keyword `abstract` can only be extracted in the context of classes through the *EClass* property *abstract*, while the modifier keyword `final` can only be extracted in the context of fields through the *EStructuralFeature* property *unchangeable*. Access level modifiers are extracted indirectly because methods and fields are extracted if their access level modifiers match the behavior of their generated Ecore counterparts. The modifier keywords `static`, `synchronized`, `volatile`, `native`, and `strictfp` are not extracted. Generics and wildcard types are extracted without problems because both are represented in the Ecore meta-metamodel. While the Ecore metamodel extraction was primarily developed for the Java code *Ecorification*, it can also be independently used as a reverse engineering approach.

5. Ecorification of Java Code

This chapter explains in detail the concept of the Java code *Ecorification*, which has the goal to achieve the automatic integration of Ecore functionality into Java code. The Java code *Ecorification* has the goal to integrate Ecore functionality into Java code while preserving all of its original functionality. Preserving the original functionality of the Java code requires retaining the interfaces which are offered by the modules of the code and also retaining all internal functionality of the code. That means the product of the Java code *Ecorification* is the original code, enriched with the desired Ecore functionality. This code then can be used with Ecore-dependent tools.

The process of the *Ecorification* of Java code can be split up into four steps: First, generating an Ecore representation of the original Java code. The Ecore representation consists of two parts: An Ecore metamodel and its generated model code. Second, generating the integration code, which will allow interlacing the original Java code and the generated Ecore model code. Third, adapting the original Java code to allow interlacing the adapted code and the Ecore model code with the help of the integration code. Fourth and last combining all three codes into the *decorated code*, which contains the implementation details of the original code and the Ecore functionality of the model code. This way, the *Ecorification* of Java code allows the integration of Ecore functionality. These Steps are depicted in Figure 5.1.

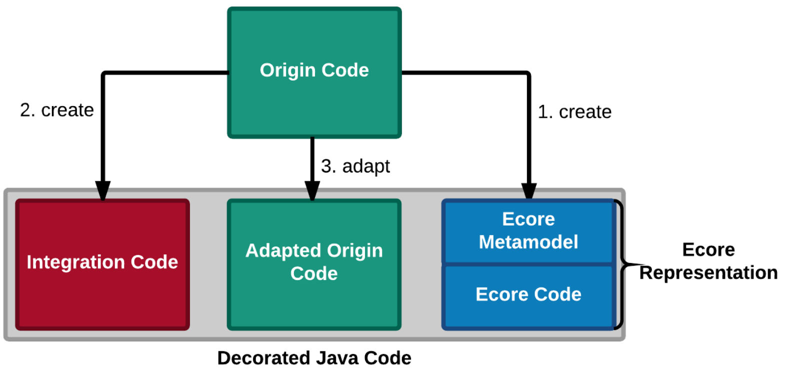


Figure 5.1.: A simple depiction of the *Ecorification* process, which is divided into four steps: The Ecore representation creation, the *integration code* generation, the adaption of the original Java code and combining the three codes.

Besides the original Java code, we address four different types of source code in the *Ecorification* process. To avoid confusion between those five different types of code, this thesis introduced the following naming scheme for the *Ecorification* of Java code:

- *Origin code* describes the original Java code, into which Ecore functionality is integrated.
- *Ecore code* describes the Ecore model code that was generated from the extracted Ecore metamodel.
- *Adapted origin code* references the *origin code* after being edited to allow the integration of Ecore functionality through the interlacing process.
- *Integration code* describes the code that is generated to interlace the *Ecore code* and the *adapted origin code*.
- *Decorated code* is the final product of the *Ecorification* of Java code, the Java code with integrated Ecore functionality.

First, Section 5.1 describes what the Ecore representation is and how it can be generated. Second, Section 5.2 explains the base concept of the Java code *Ecorification* for types. Section 5.3 then extends that concept to cover how specialization and realization relations are retained during the *Ecorification* of Java code. The method delegation of the *integration code* is outlined in 5.4. Section 5.5 explains the removal of fields of the *origin code* and the reason behind it. Last, two problems of this approach and their possible solutions will be outlined: The adaption of the Ecore factories in Section 5.6 and replacing parameterized constructors in Section 5.7.

5.1. Ecore Representation

The Ecore representation of the *origin code* is a key part of the *Ecorification* process. It consists of two parts: An Ecore metamodel and the *Ecore code*. The Ecore representation models the *origin code* and has the goal to replicate it as closely as possible. The *Ecore code* is the Ecore model code that was generated from the extracted Ecore metamodel. Contrary to the *origin code*, the Ecore representation already contains the desired Ecore functionality. However, it does not contain the implementation details of the *origin code*. For that reason, the Ecore representation has to be interlaced with the *origin code*.

The Ecore representation is generated with the help of the Ecore metamodel extraction, which was explained in Chapter 4. The Ecore metamodel extraction generates an Ecore metamodel from the *origin code*. The metamodel is then used to generate the *Ecore code*. Generating Java code from Ecore metamodels is a fundamental feature of EMF and can be accomplished programmatically using the Ecore API.

Because the *Ecore code* was generated from an Ecore metamodel, it matches the EMF generator patterns (see Chapter 10 in [31]). When generating Ecore model code from an *EPackage*, the code is organized into two Java packages. The first Java package is named after the *EPackage* itself and contains the interfaces that represent the *EClasses*. The second package is a subpackage of the first package and is named `impl`. It contains the classes that implement the interfaces of the first package. That means it contains the implementation of the *EClasses* (see Chapter 10 in [31]).

Additionally to these classes, the Eclipse Modeling Framework also generates two interfaces and their implementations for the *EPackage*: The package and the factory. While the package provides constants and convenient methods for the metadata of the *EPackage*, the factory allows creating instances of the packages interfaces (see Chapter 10.8 in [31]).

As a consequence of these EMF generator patterns, the *Ecore code* of the Ecore representation contains two classes and two interfaces per *EPackage* and one class and one interface per *EClass*. In the context of the Ecore metamodel extraction, this means that every class in the *origin code* is represented in the Ecore representation by one class and one interface. While they do not contain the implementation details of the original class, they do contain the Ecore functionality that is desired for the *Ecorification* of Java code.

Figure 5.2 depicts the *Ecore code* of the Ecore representation of the running example (see Chapter 3). The classes `Person` and `Customer` are both represented through an Ecore interface and an implementation class. The interface `BusinessPerson` in only represented through an Ecore interface. The supertype hierarchy of the running example is represented through the realization relations of the interfaces. Both the interface `EPerson` and `EBusinessPerson` extend the interface `EObject` since they do not have a super interface. While the methods are declared in the Ecore interfaces, they are implemented in the implementation classes.

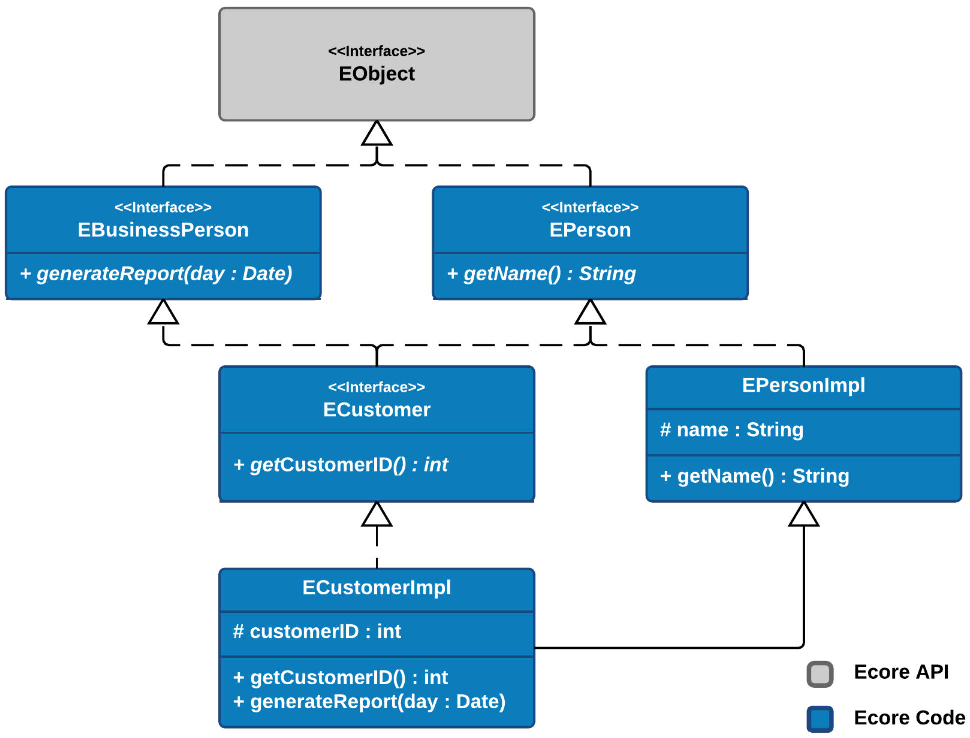


Figure 5.2.: UML class diagram that depicts the *Ecore code* of the Ecore representation of the running example (see Chapter 3).

5.2. Base Concept for Types

The integration of Ecore functionality into the *origin code* can be achieved through interlacing the *origin code* with the *Ecore code*. As described in Section 5.1, the *Ecore code* contains an interface and an implementation class for every class in the *origin code*. There are several approaches to integrate the functionality of the *Ecore code* into the *origin code*.

For example, classes from the *origin code* could simply inherit from the correlating implementation classes of the *Ecore code*. This would require changing the implementation classes to keep all functionality of the *origin code*. Another way would be that classes from the *origin code* could implement the correlating interfaces of the *Ecore code* and then delegate all unimplemented Ecore functionality to instances of the implementation classes of the *Ecore code*. These instances can be created with the factories of the packages. That would prevent the *Ecore code* from being edited, but require the *origin code* to delegate the Ecore functionality to instances of the implementation classes of the *Ecore code*.

A third approach is to use wrapper classes to connect the *Ecore code* and the *origin code*. That allows encapsulating the functionality which integrates both types of code into wrapper classes. That means the classes of the *origin code* extend wrapper classes, which implement the Ecore interfaces in the same way the classes of the *origin code* did in the previous approach. The wrapper classes delegate all unimplemented Ecore functionality to instances of the correlating implementation classes of the *Ecore code*, which they can create with the factories of the packages. This thesis chooses the last approach. To explain this approach on a simple level, we now assume the classes of the *origin code* have neither specialization realization nor realization relations. That means the *origin code* is a set of classes without any super type declarations.

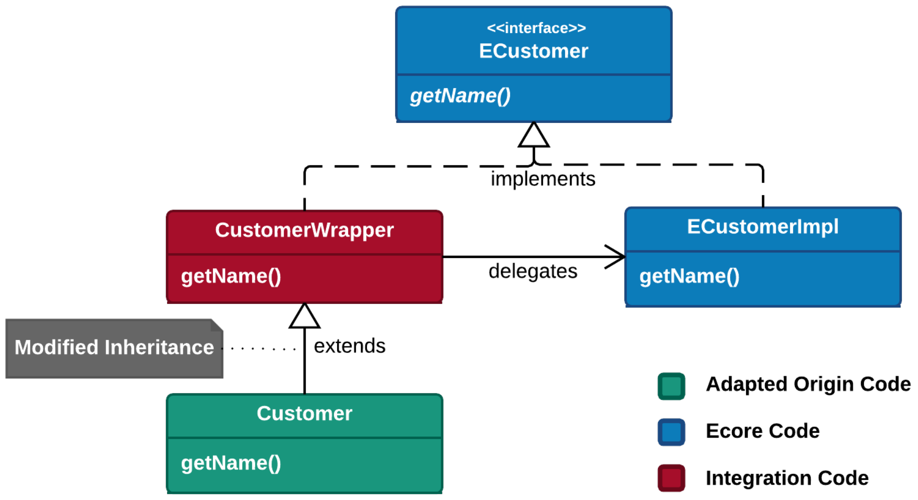


Figure 5.3.: UML class diagram of the base concept: The wrapper class `CustomerWrapper` encapsulates the functionality that is needed to connect the class from the *origin code* with the *Ecore code*.

Figure 5.3 depicts an example of this approach. The class `CustomerWrapper` implements the Ecore interface `ECustomer` and delegates functionality to the implementation class

`ECustomerImpl`. The class `Customer` extends the class `CustomerWrapper` and therefore inherits its functionality. For every class of the *origin code* we now create a wrapper class as seen in Figure 5.3. This wrapper class implements the correlating interface of the *Ecore code*. All Ecore functionality is then delegated to the implementation class of the *origin code*. The class from the *origin code* has to be edited to inherit from the wrapper class. The edited *origin code* is what we call *adapted origin code*. These wrapper classes, which connect the *Ecore code* and the *adapted origin code*, form the *integration code*. Using this approach, the *adapted origin code* inherits the Ecore functionality from the wrapper classes of the *integration code*. This means that the classes of the *adapted origin code* can be used in the same way as the *origin code* could be used before the Java code *Ecorification* process.

We now have a base concept for the interlacing of classes, which means we can create a connection between the types of the *Ecore code* and the classes of the *origin code*. However, the Java language has other important types: Interfaces and enums. While it is possible to interlace interfaces in the *origin code* with their Ecore counterparts, this thesis does not yet implement that feature. Interfaces could be interlaced by implementing their Ecore counterparts. While enums are a special form of classes, they cannot be interlaced the same way as classes. This thesis does not interlace enums with their Ecore counterparts. While this could be useful, it is not easy to implement. One problem lies in the role of *EEnums* in the Ecore meta-metamodel. As explained in Section 4.2, the functionality of an *EEnum* is very limited compared to an enum in the Java language. *EEnums* cannot have *EOperations* or *EStructuralFeatures*. That means, to create an Ecore representation for an enum, one would have to extract an *EEnum* for the enum constants and an *EClass* for the other functionality of the enum. When integrating Ecore functionality in an enum, the enum would have to be interlaced with both the *Ecore code* counterpart of the *EClass* and the *EEnum*.

5.3. Retaining Specialization and Realization Relations

In Section 5.2 we described the base concept for the Java code *Ecorification* process. For simplicity, we assumed that the classes of the *origin code* have neither specialization relations nor realization relations. This assumption cannot be made for real Java code. Therefore, we have to consider *origin code* with types highly interconnected through specialization and realization relations. That means the *Ecorification* process interferes with the specialization relations of the *origin code*.

The previous section also introduced the adaption of the *origin code*. One part of this adaption was to edit the classes of the *origin code* to let them inherit from the correlating wrapper classes of the *integration code*. However, to keep the functionality of the *origin code*, it is important to retain the interclass relations of the *origin code*. This section, therefore, discusses how to retain the original specialization and realization relations while keeping the *origin code* interlaced with the *integration code* and the *Ecore code*.

If we look at the running example (see Chapter 3), the class `Customer` extends the class `Person` and implements the interface `BusinessPerson`. In this example, the *Ecorification* process would overwrite the existing inheritance relation, so that the class `Customer` would extend the class `CustomerWrapper` from the *integration code* (See Figure 5.3). This shows the need to add additional specialization and realization relations to retain the functionality of the *origin code*.

5.3.1. Realization Relations

The *adapted origin code* adopts realization relations from the *origin code* without modifying them. That means the realization relations are not overwritten during the *Ecorification* process. However, because classes from the *adapted origin code* can be declared as types from the *Ecore code*, the interface from the *Ecore code* should extend the same interfaces that the corresponding class from the *adapted origin code* implements. In the case of the running example (see Chapter 3), this means that the corresponding *Ecore code* interface of the class `Customer` needs to extend the interface `BusinessPerson`. This is implemented in Figure 5.4, where the interface `BusinessPerson` is implemented by the class `Customer` and extended by the interface `ECustomer`.

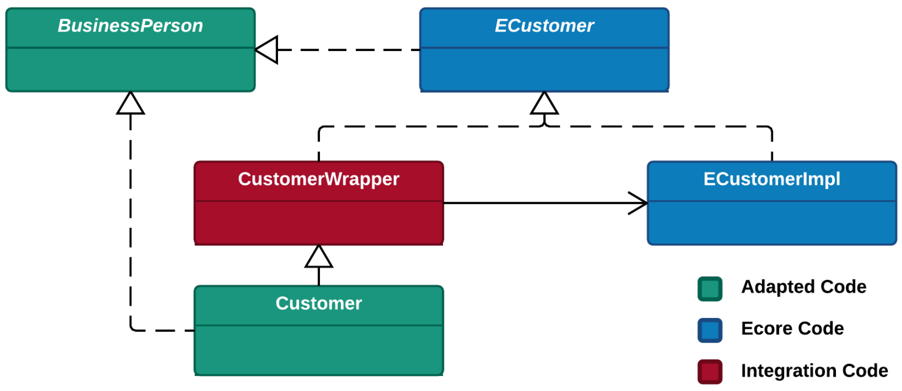


Figure 5.4.: UML class diagram of the extended base concept: Realization relations between classes.

This can also be solved by implementing the base concept of the *Ecorification* (see Section 5.2) for interfaces. Implementing that base concept for interfaces would result in the generation of an interface in the *Ecore code* for every interface in the *origin code*. The generated interfaces of the *Ecore code* are then naturally included in the realization relations of the *Ecore code*.

5.3.2. Specialization Relations

As described before, the *Ecorification* process interferes with the specialization relations of the *origin code* to allow the integration of the Ecore functionality. More specifically, the classes of the *adapted origin code* extend the wrapper classes from the *integration code*. Therefore, all existing specialization relations of the *origin code* are overwritten. To retain the original specialization relations, the wrapper classes of the *integration code* need to extend the superclasses of their correlating classes in the *origin code*.

In the running example (see Chapter 3), the `Customer` extends the class `Person`. This specialization relation is retained in Figure 5.5. The class `Customer` was changed to extend

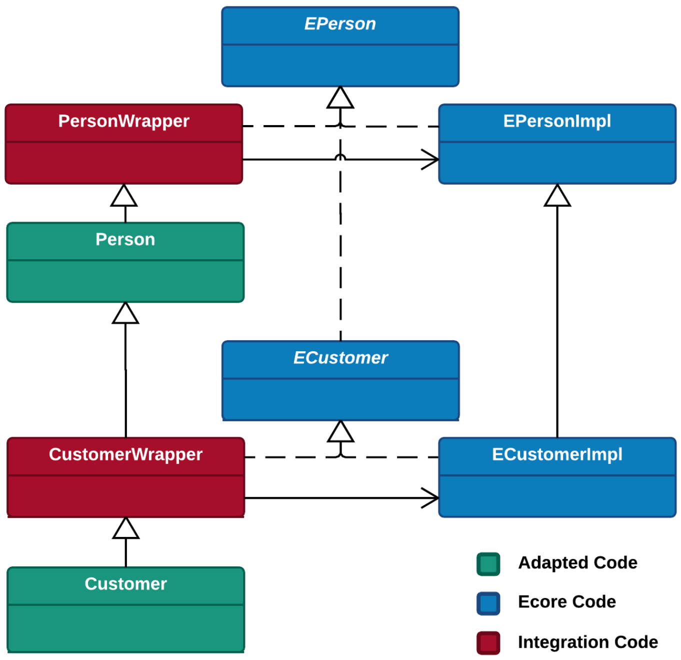


Figure 5.5.: UML class diagram of the extended base concept: Specialization relations between classes.

the wrapper class `CustomerWrapper`. However, it still inherits indirectly from the class `Person` because the wrapper class `CustomerWrapper` extends `Person`. This way the Ecore functionality is integrated without removing inherited functionality.

5.4. Method Delegation in the Integration Code

As explained before, the *integration code* consists of the wrapper classes, which connect the *Ecore code* and the *adapted origin code*. They are the centerpiece of the interlacing process. The *integration code* serves two purposes: The first one is to retain the inherited functionality of the *origin code* in the *adapted origin code* through adopting the specialization relations of the *origin code* in the correlating wrapper classes. This purpose was already discussed in Section 5.3. The second purpose is the interlacing of the *adapted origin code* and the *Ecore code* itself. Because the classes of the *adapted origin code* extend their correlating wrapper classes, they inherit all functionality of the *integration code*. The wrapper classes themselves then obtain the Ecore functionality through implementing the interfaces of the *Ecore code*. We previously mentioned that there is the need to delegate the methods that were declared by the Ecore interface. This section covers in detail how this delegation of the declared methods works.

The methods of the Ecore interfaces are delegated to the Ecore implementations classes, which implement the Ecore interfaces as well. Therefore, the wrapper classes need a reference to an instance of the implementation classes. These instances can be created through the package factories generated by EMF. For every wrapper class, we divide all methods that are declared in its Ecore interfaces (locally declared or inherited from a super interface) into three categories: First, the methods that were originally declared in the *origin code* counterpart of the interface. We call these methods local methods. An example of a local method is `getCustomerID()` in the class `Customer` of the running example (see Chapter 3). Second, the methods that were originally declared in the superclass of the *origin code* counterpart. For example the method `getName()` in the class `Person` of the running example (see Chapter 3). Third, the Ecore methods, which have no counterpart in the *origin code*. For example the method `eClass()`, which is declared in the interface `EObject`. The reason for this categorization is that each of these categories has to be treated differently when implementing the delegation.

5.4.1. Delegating Local Methods

The local methods, which were originally declared in the correlating *origin code* classes of the interfaces have to be delegated differently depending on whether they are access methods (see Section 4.4.1) or not. If they are access methods, they have to be delegated to the Ecore implementation of the interface, because the *adapted origin code* does not contain the fields of the *origin code*, which should be referenced by the access methods. This is the case because EMF creates own fields for all *EStructuralFeatures* when generating the *Ecore code* (see Chapter 10.1.2 in [31]) and all fields of the *origin code* are extracted during the Ecore metamodel extraction as *EStructuralFeatures*. As a result, the *adapted origin code* delegates all field access to the access methods of the correlating wrapper classes of the *integration code*. This delegation is depicted in Figure 5.6. The interface `ECustomer` contains an access method called `getCustomerID()`. The class `CustomerWrapper` implements that method and delegates it to the class `ECustomerImpl`.

If they are not access methods, these methods are still implemented in the *adapted origin code*. That means the classes of the *adapted origin code* do not need an implementation in the *integration code*. As a result, there are two options for the delegation process: The first option is to declare the wrapper classes of the *integration code* as abstract (see §8.1.1.1 in the JLS [12]). This means the wrappers do not have to implement the methods of their Ecore interfaces that are not access methods. The second option is to implement those and delegate to the correlating implementation of the method in the Ecore implementation class of the *Ecore code*. These methods in the Ecore implementation just throw an `UnsupportedOperationException`. This provides a dummy implementation of the method for the wrapper class while the original methods remain unchanged. The second option is implemented in the example that is depicted in Figure 5.6: The method `generateReport()` is implemented in the class `Customer`, as it was in the *origin code*. Because the Ecore interface `ECustomer` declares that method as well, the class `CustomerWrapper` delegates it to the class `ECustomerImpl`. While the first option is more elegant, the second implementation can be useful when utilizing delegation patterns such as the active annotation `@delegate` of the Xtend language, which treats all delegated methods the same way.

5.4.2. Delegating Inherited Methods

Methods whose signatures are implicitly declared by the Ecore interfaces of the *Ecore code* through inheriting them from their super interfaces are not part of the correlating class

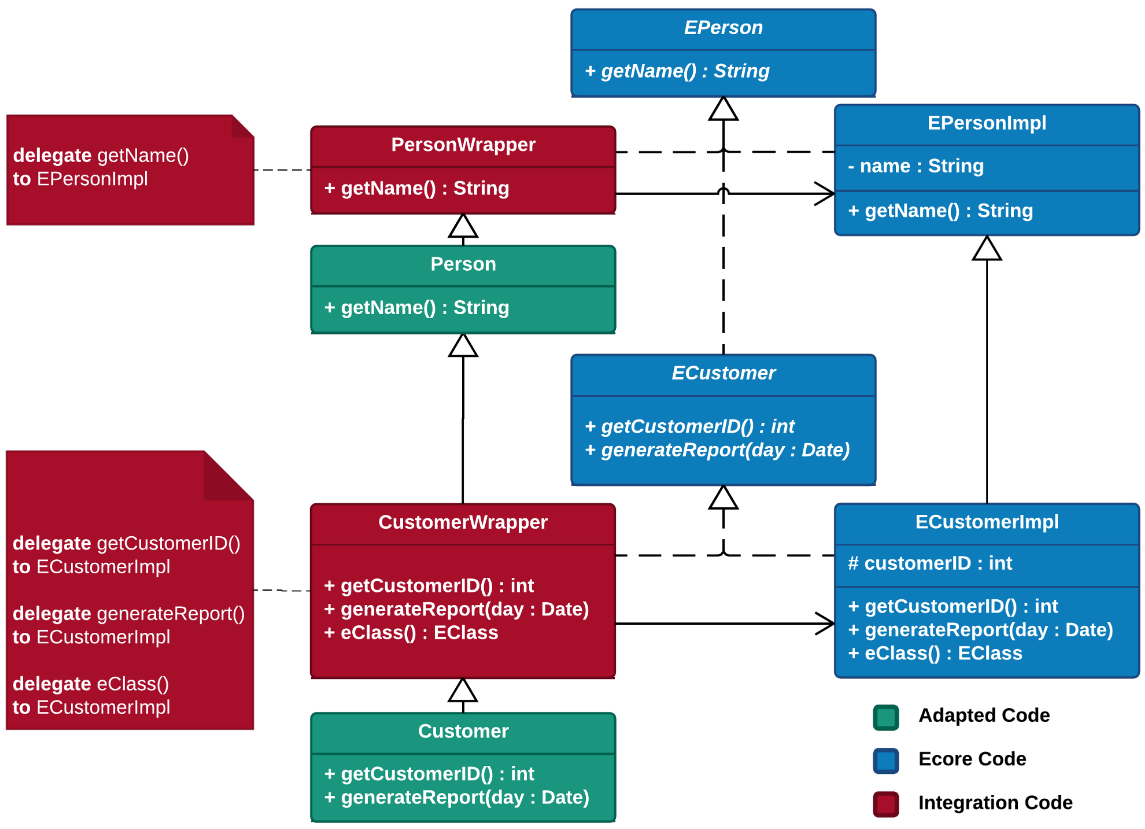


Figure 5.6.: UML class diagram of the delegation process of the *integration code*.

of the *origin code*. The correlating classes of the *origin code* inherit them through their specialization relations. The wrapper classes must not delegate these methods to the Ecore implementations because they are implemented in the original superclasses. That means the wrapper classes inherit the implementation of these methods from their super classes, which were originally the superclasses of the *origin code* (See Section 5.3).

This behavior is depicted in Figure 5.6. The class `Person` contains the implementation of the method `getName()`. The Ecore interface `EPerson` declares the signature of that method. The class `CustomerWrapper` and therefore also the class `Customer` inherit the declaration from the interface `EPerson` as well as the implementation of the class `Person`.

5.4.3. Delegating Ecore Methods

Ecore methods, which have no counterpart in the *origin code*, are created when generating code from an Ecore metamodel. The signatures of these methods are declared in the interface `EObject`. They are a part of all Ecore interfaces of the *Ecore code* because the root interface of the Ecore interface hierarchy is the interfaces `EObject`.

These methods are implemented in the implementation classes of the *Ecore code*. The wrapper classes of the *origin code* delegate these methods to the implementation classes. As a result, the *adapted origin code* inherits these Ecore methods and their functionality.

This can be seen in Figure 5.6, where the class `CustomerWrapper` delegates the method `eClass()` to the class `ECustomerImpl`.

5.5. Removing the Fields of the Origin Code

During the *Ecorification* process, the classes of the *origin code* will be adapted to allow the integration of Ecore functionality. This turns the *origin code* into the *adapted origin code*. This happens in two steps: First, the superclass declaration will be overwritten. Second, the fields of the classes are removed. While the first step was already explained in 5.2, the second step was only mentioned, but not explained. This section explains the removal of the fields in detail.

The second step, removing the fields from the classes is necessary because the Ecore representation already contains the same fields and access methods in the implementation classes of the *Ecore code*. They are automatically created when generating the *Ecore code* from the extracted Ecore metamodel. First, we replace all references to the fields of the classes in the *origin code* by equivalent access methods calls. Then, we remove the fields and their access methods. All access method calls of this class now are executed by the wrapper classes of the *integration code*. As discussed in Section 5.4, they are then delegated to the correspondent implementation classes of the *Ecore code*. As a result of this adaption, the classes of the *adapted origin code* will inherit the Ecore functionality from the wrapper classes of the *integration code*. Also, while the implementations of the methods remain in the *adapted origin code*, the fields are part of the *Ecore code*.

There is one problem with this approach: If the access methods of the *origin code* contain more functionality than just to get or set a field, this functionality is lost during the *Ecorification* process. The problem lies in the Ecore metamodel extraction. As discussed in Section 4.4, an access method generated by EMF from the extracted Ecore metamodel will differ from the original access method because the generated access methods contain no additional functionality. As a result, the access methods with the additional functionality of the *origin code* will be removed, and all calls to them will be delegated to the basic access methods of the *Ecore code*. This could be avoided by implementing a strategy to retain the code of the access methods.

One strategy that solves that problem is defining a custom *EAnnotation* during then Ecore metamodel extraction. This annotation then contains the source code of the access methods in a similar way as the *EAnnotation* used by EMF to contain the source code of *EOperation* bodies. The custom *EAnnotation* can then be attached to the *EStructuralFeatures* that represent the fields referenced by the access methods. During the *Ecorification* process, this annotation could be used to add the missing additional functionality to the access methods.

5.6. Changing the Factories

As mentioned in Section 5.1, EMF creates a package and a factory class for every *EPackage* when generating model code from an Ecore metamodel. These factories allow creating instances of the types contained in the package. Every factory consists of an interface and an implementation class. While it is not necessary to use the factories for the instantiation of the types in the package, EMF highly encourages it (see Chapter 10.8 in [31]). Additionally, many tools like model transformation languages use these factories

While the previous sections covered the interlacing of the *origin code* and the *Ecore code* with the help of the *integration code*, the factories are still creating instances of the implementation classes of the *origin code*. If we want to use our *adapted origin code* as we would normal model code generated from Ecore metamodels, the factories have to create instances of the classes from the *adapted origin code*. To achieve that, we have to create adapted factories. It is important to keep the original factories as well because they are needed to create instances of the *Ecore code* implementation classes for the *integration code*. As discussed in Section 5.4, the wrapper classes of the *integration code* delegate certain methods to the implementation classes of the *Ecore code*.

To keep the original factories intact, we copy them into a subpackage of the factory package. While the copies are not being altered, we change the original factories to create *adapted origin code* instances. We adapt the methods in the implementation classes of the factories to return the correlating class of the *adapted origin code* instead of the *Ecore code* implementation class. The method signature of the implementation class and the interface does not have to be changed because both the classes from the *adapted origin code* and the *Ecore code* implement the interfaces of the *Ecore code*. For the *Ecore code* class `ECustomer` of the running example (see Chapter 3) the original instantiation method in the factory class looks like this:

```
public ECustomer createECustomer() {
    ECustomerImpl customer = new ECustomerImpl();
    return customer;
}
```

An instance of the class `ECustomerImpl` is created and returned. The method declares the return type `ECustomer`. This can be easily altered to create an instance of the class `Customer` of the *adapted origin code* by changing one single line in the method body:

```
public ECustomer createECustomer() {
    Customer customer = new Customer();
    return customer;
}
```

While the declared return types stay the same, the actual return type is changed. After copying the factories and adapting the original ones the *decorated code* can be used as one would normal model code. While the interfaces of the adapted factories do not show any change, internally, classes of the *adapted origin code* are instantiated.

5.7. Replacing Parameterized Constructors

As mentioned in Section 4.4, The Ecore meta-metamodel does not use constructors or any equal counterpart. The reason for this is that constructors are not necessary for the Ecore metamodel and automatically generated in the model code. Initial values of *EStructuralFeatures* can be declared with default value properties (see Chapter 5.3 in [31]). The model code generated from Ecore metamodels only uses default constructors to create the classes that represent the *EClasses* of the metamodel (see Chapter 10.8 in [31]). Initial values of fields are set through static default value fields in the same classes (see Chapter 10.2 and 10.3 in [31]).

Section 4.4 explained that constructors can only be extracted as normal *EOperations* or cannot be extracted at all. Not extracting parameterless constructors causes no problems because the Ecore factories call the public parameterless constructor of the classes they create (see Section 5.6). In contrast, extracting parameterized constructors will lead to problems. For example, if a class only offers a parameterized constructor, the instantiation of this class depends on the constructor parameters. However, because the Ecore interfaces of the factories do not allow to parameterize the creation methods, these methods can only create one specific instance type of a class with a specific set of parameters.

If we look at the factory example of the previous section and assume the class `Customer` only offers a constructor with two parameters `name` and `customerID`, the `Customer` creation method `createECustomer()` of the factory would have to be edited to call this parameterized constructor. However, as a result, the method `createECustomer()` then only allows creating one customer with a fixed `name` and `customerID` because the factory interface only defines the method signature of `createECustomer()` without any parameters. For example a customer with the `name` "John Doe" and the `customerID` "17701362" in the following method:

```
public ECustomer createECustomer() {
    Customer customer = new Customer("John Doe", 17701362);
    return customer;
}
```

Relevant for the *Ecorification* process are the initial values of fields, which are set with parameterized constructors.

To solve this problem, classes that depend on the instantiation with parameters need a parameterless constructor and a special method for the initialization process that depends on the parameters of the constructor. The availability of a parameterless constructor can be achieved by manually creating a public parameterless constructor. The method for the initialization process needs to take the same parameters as the parameterized constructor and should contain all the parameter dependent code of the parameterized constructor. This method then can be used by the user after the instance was created with an Ecore factory method. This solution still allows existing code to use the parameterized constructors. Additionally, the solution offers an initialization method as a workaround to execute the parameter dependent parts of the parameterized constructors after the instance was created with the factories. In our previous example, the class `Customer` would now have two additional methods:

```
public Customer() {
    // parameter independent code.
}
```

```
    public void initialize(String name, int customerID) {
        // parameter dependent code:
        this.name = name;
        this.customerID = customerID;
    }
```

A parameterless constructor and the method `initialize()`. The parameterless constructor gets called by the factory method `createECustomer()` and the method `initialize()` can be called after the object creation.

6. Prototypical Implementation

The Java code *Ecorification* concept was implemented in two parts: First, Ecore metamodel extraction was implemented. Second, the extraction implementation was used to implement a prototype of the Java code *Ecorification*. The separated implementation allows using the Ecore metamodel extraction for other purposes than the integration of Ecore functionality into Java code. While the Ecore metamodel extraction implementation covers nearly all features of Section 4, the Java code *Ecorification* prototype is only implemented as proof of concept. The reason for this is the time limit of the thesis. Both the Ecore metamodel extraction and the Java code *Ecorification* are implemented as Eclipse plugins using the Java Development Tools and the Ecore API. While the Ecore metamodel extraction was solely implemented with the Java Language, the Java code *Ecorification* was implemented with the Java Language and Xtend, a programming language for the Java Virtual Machine. Both together, the Ecore metamodel extraction and Java code *Ecorification* were implemented in under 3200 lines of code. The code for both the Ecore metamodel extraction[1] and the Java code *Ecorification*[2] is publicly available under the Eclipse Public License [9]. First, Section 6.1 covers the extraction implementation. Second, Section 6.2 covers the *Ecorification* implementation.

6.1. Extraction Implementation

The extraction implementation is an Eclipse plugin that extracts Ecore metamodels from Java projects in the Eclipse workspace. It supports the extraction of packages with their hierarchy, interfaces, classes, enumerations with their enum constants, inheritance and realization relations, fields, methods with their parameters, return types and throws clauses, generic types with their type parameters, generic arguments, and wildcard types.

The extraction implementation uses an internal metamodel called *Intermediate Model* during the extraction process. This metamodel extends the model element mapping defined in Chapter 4. First, Section 6.1.1 introduces the general architecture of the extraction implementation. Second, Section 6.1.2 explains the *Intermediate Model* in detail. Third, Section 6.1.3 shows the extended model element mapping that was used in the extraction implementation. Fourth and last, 6.1.4 explains how the extraction process of the extraction implementation can be configured.

[1] github.com/tsaglam/EcoreMetamodelExtraction
[2] github.com/tsaglam/JavaCodeEcorification

6.1.1. Architecture

The general architecture of the project can be broken down into three modules: The Java project extractor, the *Intermediate Model*, and the Ecore metamodel generator. These three modules encapsulate different steps of the extraction workflow (see Figure 6.1). The Java project extractor analyzes a Java project using the JDT API and extracts all the information the implicit metamodel of the Java code contains. With this information, the extractor then builds an instance of the *Intermediate Model*. The *Intermediate Model* is an internal metamodel that serves as an interim stage between the Java code and the Ecore metamodel. The Ecore metamodel generator uses the Ecore API to build an Ecore metamodel from the *Intermediate Model* instance. Additionally, the Ecore metamodel generator contains saving strategies that save the generated Ecore metamodel as an Ecore file.

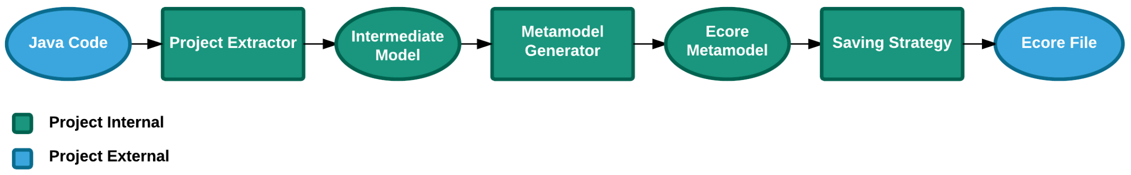


Figure 6.1.: The depiction of the Ecore metamodel extraction workflow. Modules are rectangular, and products are oval. The project Java project extractor extracts an *Intermediate Model* from the Java code, which is then used by the Ecore metamodel generator to generate an Ecore metamodel. This metamodel can be saved into an Ecore file with the help of the saving strategies are a submodule of the Ecore metamodel generator.

The use of the *Intermediate Model* as an interim stage during the extraction process has several benefits. First, it allows the separation of the extractor module and the generator module. As a result, both the Ecore metamodel generator and the Java project extractor are replaceable. For example replacing the Java project extractor with a different module that extracts an *Intermediate Model* from another programming language, allows the extraction of Ecore metamodels from that programming language. Second, the use of the *Intermediate Model* allows to conveniently select which parts of the Java code should be extracted into the Ecore metamodel.

6.1.2. Intermediate Model

The *Intermediate Model* is a metamodel that uses a tree structure that contains all extracted information of a Java project. It tries to stay close to the implicit model of the language Java and does not reflect the characteristic properties of the Ecore meta-metamodel. Through its tree hierarchy, it is possible to deselect subtrees of the *Intermediate Model* for the metamodel generation. When an *Intermediate Model* gets created by the project extractor, all extracted information is added to the *Intermediate Model*, no matter if that information can be represented in an Ecore metamodel or not. That means an *Intermediate Model* contains equally as much or more information than an Ecore metamodel generated from that *Intermediate Model*. The tree structure of an *Intermediate Model* instance starts with an `ExtractedPackage` as root package, which contains all other model elements. The *Intermediate Model* can be divided into two different hierarchies. The element hierarchy and the datatype hierarchy.

The *Intermediate Model* representations of packages, classes, interfaces, enumerations, and methods are instances of the class `ExtractedElement`. As a result, they are part of the element hierarchy (see figure 6.2). An `ExtractedPackage` contains any number of instances of `ExtractedPackage` to represent subpackages. It also contains any number of instances of `ExtractedType` to represent the interfaces, classes, and enums of the package. An `ExtractedType` has instances of the classes `ExtractedMethod` and `ExtractedField`. These are the methods and fields of the type. An `ExtractedEnum` additionally has enum constants, which are instances of the class `ExtractedEnum constant`. An `ExtractedMethod` contains an `AccessLevelModifier`, a `MethodType` as well as instances of the classes `ExtractedParameter` and `ExtractedDataType`. The instances of the class `ExtractedParameter` represent method parameters, while the instances of the *Intermediate Model* class `ExtractedDataType` represent the method return type and the exception declarations of the throws clause.

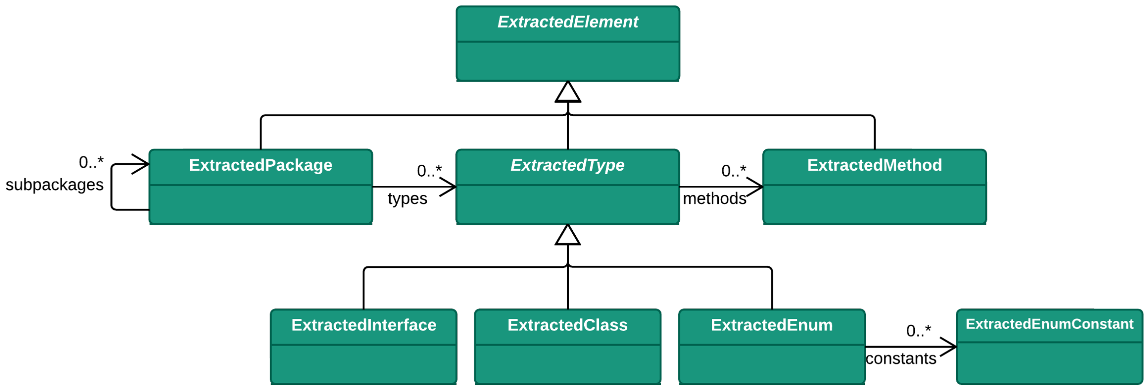


Figure 6.2.: UML class diagram of the element hierarchy, a subset of the *Intermediate Model*.

The *Intermediate Model* elements that represent data types, parameters, variables, and fields are all part of the data type hierarchy (see Figure 6.3). They are connected through inheritance. A data type, modeled through the class `ExtractedDataType` represents a single data type. An `ExtractedVariable` is an `ExtractedDataType` with a variable name, also called identifier. A parameter, modeled through the class `ExtractedParameter` represents a method parameter. A field, modeled through the class `ExtractedField`, represents a field of a type. Both classes `ExtractedParameter` and `ExtractedField` extend the class `ExtractedVariable`. One difference between a field and a variable is that the field has an access level modifier, represented through the enum `AccessLevelModifier`. A generic type parameter is modeled through the class `ExtractedTypeParameter`. The type parameter has any number of type parameter bounds. The bounds are represented by a list of `ExtractedDataType` instances.

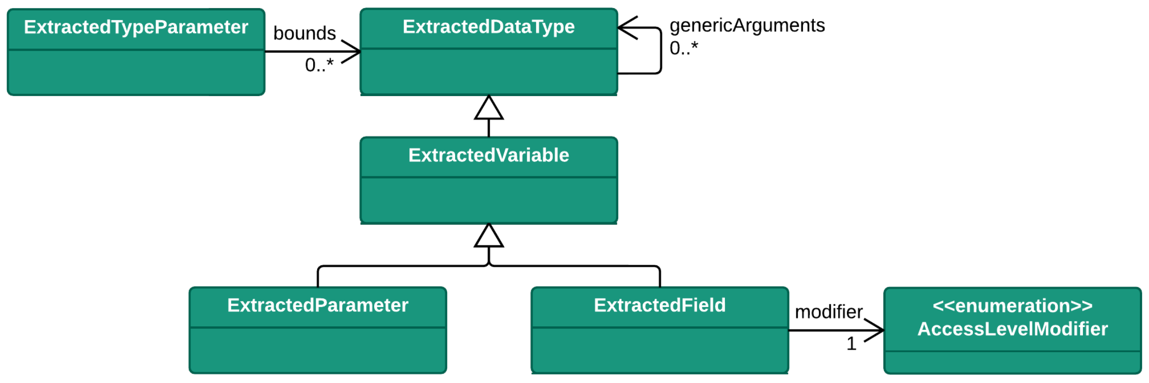


Figure 6.3.: UML class diagram of the data type hierarchy, a subset of the *Intermediate Model*.

6.1.3. Extraction Table

As explained in Section 6.1.1, Ecore metamodel extraction uses the *Intermediate Model* between the extraction of information from the Java code and the generation of the Ecore metamodel. It uses a mapping from Java elements to *Intermediate Model* elements and *Intermediate Model* elements to Ecore elements. Therefore, the mapping between the elements of the Ecore meta-metamodel and the implicit Java metamodel (see Table 4.1) can be extended to represent the additional step with the *Intermediate Model*. This extended mapping is visualized in the extraction table (see Table 6.1). The left column of the extraction table contains the Java elements. The right column contains the Ecore elements which are generated from the extracted features. The middle column contains the relating elements of the *Intermediate Model*.

6.1.4. Configurability

The extraction implementation is highly configurable. The plugin uses a property file to allow the configuration of the extraction process. It is possible to disable and enable the extraction of nested types, classes, interfaces, enums, custom exception and error classes, constructors, methods, and fields. For fields and methods, it is also possible to disable and enable the extraction for specific access level modifiers. The user can also customize the default package name, the data type package name, and the package suffix of nested types. When saving an extracted Ecore metamodel, there are five different saving strategies available: Saving into the original project where the metamodel was extracted from, saving into a copy of the original project, saving into a newly created project, saving into a specific project that already exists, and saving to any specified location.

6.2. Ecorification Implementation

The Java code *Ecorification* is prototypically implemented as an Eclipse plugin. It integrates Ecore functionality into the Java code of Java projects in the Eclipse workspace. While

Java Metamodel (implicit)	Intermediate Model	Ecore Meta-Metamodel
Package	ExtractedPackage	EPackage
Type	ExtractedType	EClassifier
Class	ExtractedClass	EClass
Interface	ExtractedInterface	EClass
Enum	ExtractedEnum	EEnum
Enum Constant	ExtractedEnumConstant	EEnumLiteral
Field	ExtractedField	EStructuralFeature
Method	ExtractedMethod	EOperation
Method Parameter	ExtractedParameter	EParameter
Method Return Type	ExtractedDataType	*EClassifier/EGenericType*
Throws Clause	ExtractedDataType	*EClassifier/EGenericType*
Generic Type Parameter	ExtractedTypeParameter	ETypeParameter
Generic Type Argument	ExtractedDataType	EGenericType
Generic Type Bound	ExtractedDataType	EGenericType
Super Type Reference	ExtractedDataType	*EClass & EGenericType*

Table 6.1.: Extraction table for the Ecore meta-metamodel, *Intermediate Model*, and the implicit Java metamodel. A cursive Ecore element means the corresponding Java element is represented through a reference to the cursive Ecore element.

the extraction implementation included most of the features discussed in the concept, the *Ecorification* implementation is limited to the *Ecorification* of few selected features.

Implemented features are the creation of an Ecore representation (see Section 5.1), the creation of the *integration code* (see Section 5.4), and the adaption of the superclass declarations of the *origin code* (see Section 5.5). Not implemented are the removal of the fields from the *adapted origin code* (see Section 5.5), the adaption of the Ecore factories (see Section 5.6) and a solution for the constructor problem (see Section 5.7). The *Ecorification* implementation serves as proof of concept for the concept for the integration of Ecore functionality into Java code.

It can be broken down into five modules: The extraction implementation, the *Generator Model* generator, the Model code generator, the wrapper generator and the inheritance manipulator. These five modules encapsulate different steps of the *Ecorification* workflow of the *Ecorification* implementation (see Figure 6.4).

The *Ecorification* workflow works as follows: First, the extraction implementation extracts an Ecore metamodel from a Java project. Second, the *Generator Model* generator generates an Ecore *Generator Model* with the help of the Ecore API. The *Generator Model* is an Ecore model that contains additional information for the code generation from an Ecore metamodel (see Chapter 2.4.4 in [31]). This *Generator Model* is used in the third step by the Model code generator to generate the model code for the Ecore metamodel, also using the Ecore API. As a fourth step, the wrapper generator generates the wrapper classes using Xtend templates and the Eclipse API. The wrappers classes are written in Xtend and utilize the active annotation `@delegate`. This active annotation allows the delegation

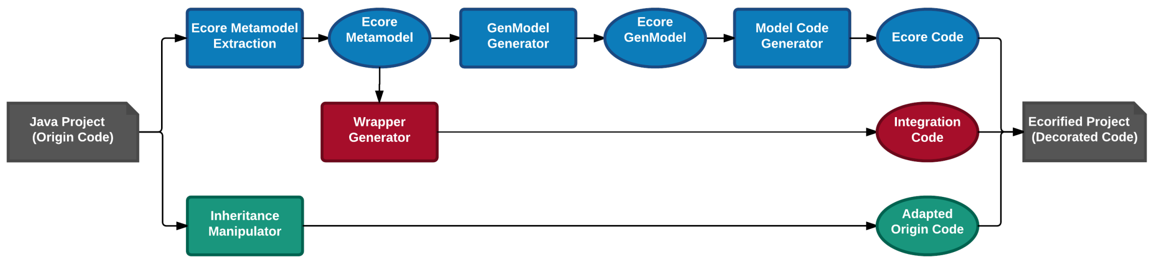


Figure 6.4.: The depiction of the *Ecorification* workflow with its five modules: The Ecore metamodel extraction, the *Generator Model* generator, the Model code generator, the wrapper generator and the inheritance manipulator. Products are depicted as ovals.

of the methods declared by the Ecore interfaces of the *origin code*. Additionally to the wrapper class generation, the necessary Xtend dependencies are added to the project. Last, the inheritance manipulator uses the Java AST of the Java Development Tools API to overwrite the superclass declarations of the classes in the *origin code* to let them inherit from the wrapper classes.

7. Validation

Because of the separation of both the concepts and the implementations of the Ecore metamodel extraction and Java code *Ecorification*, the validation was also separated. Initially, the Ecore metamodel extraction was validated by itself. After that, the whole *Ecorification* was validated, which implicitly validates the Ecore metamodel extraction again because it is used during the *Ecorification*.

First, Section 7.1 explains how the Ecore metamodel extraction was validated individually. Second, Section 7.2 describes how the Java code *Ecorification* as a whole was validated. Last, Section 7.3 discusses the problems that arose during the thesis.

7.1. Validating the Ecore Metamodel Extraction

Because there are, to our knowledge, currently no other approaches for the extraction of Ecore metamodels from Java code, it is not possible to compare the extraction implementation with other approaches. As a result, it is hard to validate, whether our result is correct. However, we used several indicators and several tests to ensure the validity of the Ecore metamodels that the extraction implementation generates. We employ three indicators for the validity of the Ecore metamodel extraction:

- Indicator 1: The validity of the extracted metamodel according to EMF. It is possible to create invalid Ecore metamodels, especially when dynamically creating metamodels with the Ecore API. For example, the Ecore meta-metamodel is designed to use *EReferences* for referring to *EClasses* and *EAttributes* for referring to *EDataTypes*. But because both *EAttributes* and *EReferences* are *EStructuralFeatures*, the Ecore API allows using an *EAttribute* to reference *EClasses*. This leads to an invalid metamodel and is not even possible when creating it with the EMF user interface. An Ecore metamodel is valid according to Indicator 1 if EMF accepts it as valid without indicating any errors.

- Indicator 2: The Ability to generate Ecore model code from the extracted metamodel. The requirements on Ecore metamodels for generating valid model code are even stricter than the requirements for the validity of the metamodel itself. That means a valid Ecore metamodel can be used to generate invalid Ecore model code. For example, an *EOperation* can reference an *EClass* that does contain any super relation that resembles the inheritance from `java.lang.Throwable` or one of its subclasses. A metamodel with this *EOperations* is valid according to Indicator 1. However, when generating model code from that metamodel, it will lead to the generation of invalid Java code. The reason for this is that a throws clause of a method can only reference subtypes of `java.lang.Throwable`. An Ecore metamodel is valid according to Indicator 2 if EMF allows the creation of a *Generator Model* without indicating any errors.

- Indicator 3: Indication through examined model properties, which can be compared to the Java code used for the extraction. This type of validation is often used to validate UML model reverse engineering approaches [18]. An Ecore metamodel is valid according to Indicator 3 if the extracted metamodel has the same values for examined model properties as the relating property of the implicit model of the Java code. We make the following model comparisons: The number of *EClasses* compared to the number of classes, the number of *EClasses* compared to the number of enums, the number of *EStructuralFeatures* per *EClass* compared to the number of fields per class, and the number of *EOperations* per *EClass* compared to the number of methods per class.

Different tests were conducted throughout the duration of this thesis. These tests utilized the different indicators to test the Ecore metamodel extraction. First, during the development of the extraction implementation, automated unit tests were used to test the different extracted features. These features were validated with Indicator 3. These unit cannot test whether the features were extracted correctly under all circumstances, they rather tested the most common behavior.

Second, a Java test project was specifically designed to test the extraction implementation. While this test project does not represent a real software project, it contains all features of the Java language that are extracted in the Ecore metamodel extraction concept. For example, it contains a package called `modifiers` that contains four types: Two classes, one interface, and one enum. One of the classes is abstract, the other is not. Each of these types contains methods and fields for every possible modifier. This package is designed to test the extraction of methods and fields for specific modifiers. This test project is publicly available[3] under the Eclipse Public License [9]. The Ecore metamodel extracted from the test project is validated with Indicator 1, Indicator 2 and Indicator 3. Indicator 3 was checked by manually counting the correlating Java elements of the test project. The extraction of the test project was successfully validated for all three indicators.

Last, two larger well-known projects were used to extract an Ecore metamodel. One is the Java AST, which a subset of the Java Development Tools (see Section 2.5). The other is the Apache Commons IO library [26] which offers utilities for input/output functionality. These projects are, opposed to the custom test project, real programs and therefore give insight into the practical application of the Ecore metamodel extraction. Table 7.1 shows the dimensions of the Ecore metamodels that were extracted from these projects. It also shows the size of the three projects in lines of code. These extracted metamodels were tested on validity with Indicator 1 and Indicator 2. Both indicators suggested the validity of the Ecore metamodel extraction. Nevertheless, it is important to acknowledge that the validity of the Ecore metamodel extraction is not guaranteed until further, more comprehensive testing.

[3]github.com/tsaglam/EME-TestProject

Element	Test Project 291 SLOC	Java AST 33.765 SLOC	Commons IO 9.957 SLOC
EPackages	18	55	29
EClassifiers	52	326	165
EClasses	29	194	118
EDataTypes	23	132	47
EEnums	2	0	1
EOperations	29	2952	402
EStructuralFeatures	24	440	192
EAttributes	21	258	186
EReferences	3	182	6

Table 7.1.: Extracted Ecore model elements from the custom test project, the Java AST, and the Apache Commons IO library.

7.2. Validating the Ecorification of Java Code

As previously explained, the Java code *Ecorification* was implemented as a proof of concept. We validated the *Ecorification* implementation through performing a QVTO transformation [13] on Java code, into which Ecore functionality was integrated through the *Ecorification* of Java code. Because QVTO transformations cannot be performed on normal Java code, this validation proofed that we were able to successfully integrate Ecore functionality into Java code. Because the adaption of the factories (see Section 5.6) was not yet implemented, we initiated the QVTO transformation programmatically. Using the QVTO user interface instead of programmatically initiating the QVTO transformation would have required the factories to create instances of the *adapted origin code* instead of the *Ecore code*.

First, two Java projects were created. The Java projects resemble a well-known example for metamodels: The "Families to Persons" example [35]. The first project, *Families* contains three classes: `Family`, `Member`, and `FamilyContainer` (see Figure 7.1). The class `Member` represents a family member and has an attribute for the first name and a reference to a family. The class `Family` represents a family. It has several references to `Member`: One father reference, one mother reference, and any number of referenced sons and daughters. The class `Family` also contains an attribute for the last name. The class `FamilyContainer` simply references any number of `Family` instances. Additionally, we added one method to the class *Family*, which prints the content of the family.

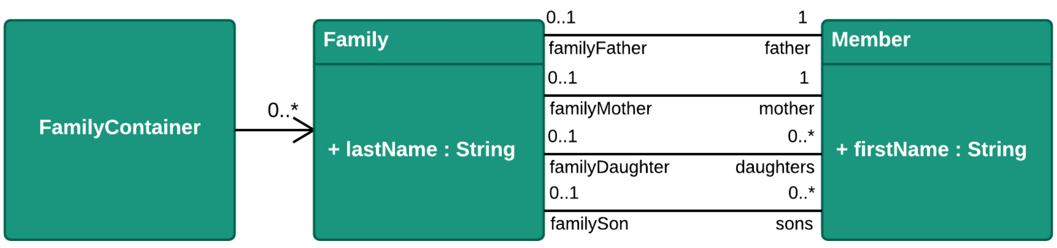


Figure 7.1.: The UML class diagram of the *Families* project.

The second project, *Persons* contains four classes: `Person`, `Male`, `Female`, and the class `PersonContainer` (see Figure 7.2). The class `Person` has an attribute for the full name of the person it represents. The classes `Male` and `Female` simply inherit from `Person`. Like the class `FamilyContainer`, the class `PersonContainer` simply serves as a reference container. It references any number of `Person` instances.

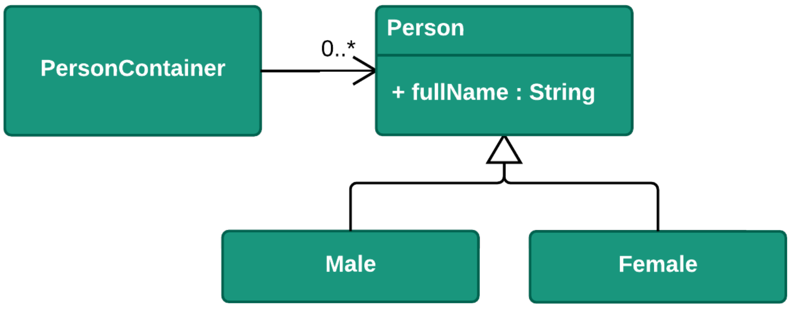
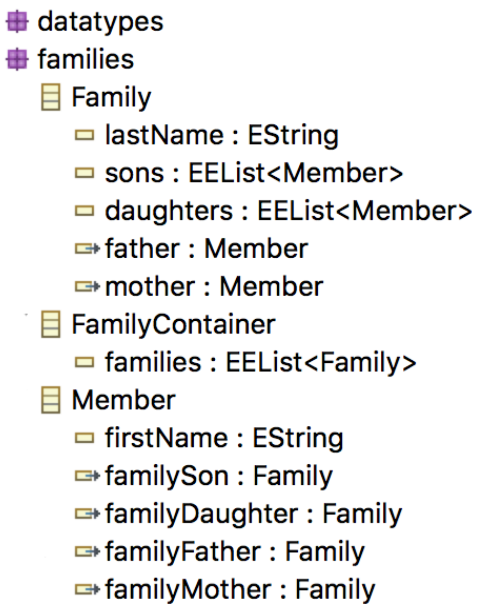


Figure 7.2.: The UML class diagram of the *Persons* project.

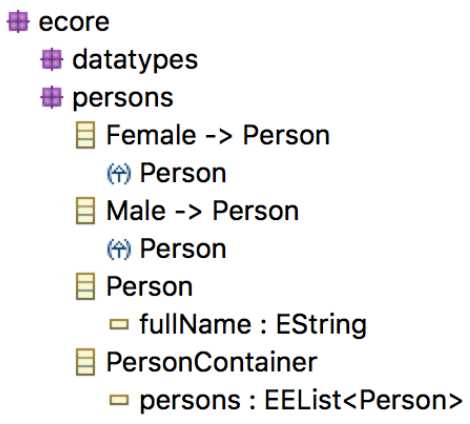
Second, we used the Java code *Ecorification* to integrate Ecore functionality into the projects *Families* and *Persons*. Each project then contained its *decorated code* and the extracted Ecore metamodel, which was generated during the Ecore metamodel extraction. The two metamodels of the projects *Families* and *Persons* can be seen in Figure 7.3 and Figure 7.4.



```
datatypes
families
  Family
    lastName : EString
    sons : EEList<Member>
    daughters : EEList<Member>
    father : Member
    mother : Member
  FamilyContainer
    families : EEList<Family>
  Member
    firstName : EString
    familySon : Family
    familyDaughter : Family
    familyFather : Family
    familyMother : Family
```

Figure 7.3.: The extracted Ecore metamodel of the Families project.

As a third step, we wrote a QVTO transformation, which transforms instances of the *Families* metamodel into instances of the *Persons* metamodel. The QVTO transformation transforms a `FamilyContainer` into a `PersonContainer` by creating `Male` instances from the `Member` instances father and sons, as well as `Female` instances from the `Member` instances



```
ecore
  datatypes
  persons
    Female -> Person
      Person
    Male -> Person
      Person
    Person
      fullName : EString
    PersonContainer
      persons : EEList<Person>
```

Figure 7.4.: The extracted Ecore metamodel of the Persons project.

mother and daughters. After the transformation, the full name of the `Person` corresponds to the first name of the corresponding `Member` and the last name of the corresponding `Family`.

We used this QVTO transformation on a simple `FamilyContainer` containing one `Family`. The `Family` has the last name "Doe" and references four `Member` instances: The father "Jon," the mother "Claire," the daughter "Alice" and the son "Bob." These roles were referenced according to Figure 7.1. After the transformation, we received one `PersonContainer` which referenced four `Person` instances: The instances of `Female` called "Claire Doe" and "Alice Doe," as well as the instances of `Male` called "John Doe" and "Bob Doe." Furthermore, we were able to call the additional print method of the class `Family` during the transformation. This method showed the same behavior as it did in the original project. This means its functionality was retained during the *Ecorification*.

The success of the QVTO transformation indicates that the Java code *Ecorification* integrated enough Ecore functionality into the projects *Families* and *Persons* to allow the application of an Ecore-dependent tool on arbitrary Java code. As a result, we conclude that the Java code *Ecorification* prototype accomplished its goal. While this did not validate the entire *Ecorification* concept with all its features, we proved the basic feasibility of the *Ecorification* of Java code.

7.3. Lessons Learned

During the validation of the Java code *Ecorification*, we discovered a problem with the *integration code*. The implementation classes of the *Ecore code* extend the static class `Container`, which is declared as a nested class in the class `MinimalEObjectImpl`. This specialization relation causes the inheritance of crucial functionality, which is needed when using the classes of the *adapted origin code* in a situation where an implementation of the Ecore interfaces is expected. As a result, our wrapper classes of the *integration code* need to extend the class `Container` as well. For the validation, we manually implemented these relations.

Additionally to this first problem, we discovered a problem with the *Ecorification* of fields. When extracting a field of type `java.util.List` from Java code, it gets generated in the *Ecore code* as a field of type `EList`. As a result, the generated accessor method will return an instance of the type `EList`. The accessor method in the *adapted origin code* will return an instance of type `java.util.List`. Because of the specialization relations, this is an occurrence of contravariance, which is not allowed in the language Java. This can be avoided by implementing the removal of the fields (see Section 5.5). However, this problem has to be solved for parameterized access methods, because they are not removed from the *adapted origin code*. For the validation, we circumvented that problem by changing the accessor method declarations in the interfaces of the *Ecore code* to declare the return type `java.util.List`.

8. Related Work

The Ecore metamodel extraction is a reverse engineering approach that extracts Ecore metamodels from Java code. While there are, to our knowledge, no other approaches that have this goal, there are many different approaches to the reverse engineering of object-oriented code. Because UML is an important tool in software engineering, many of these approaches try to extract UML models. Whereas some of the approaches describe general procedures, independent from the programming language and the model type, others are specifically tailored towards individual tools and languages. Because of the amount of reverse engineering approach, we will only name the most related and the most common ones.

"MoDisco: A Generic And Extensible Framework For Model Driven Reverse Engineering" [3] introduces the tool MoDisco, which is an Eclipse plug-in that allows reverse engineering models from existing applications. As input, it can use Java source code, databases, and configuration files. It utilizes the Java Abstract Syntax Tree of the Eclipse Java Development Tools (see Section 2.5). All extracted models are instances of the MoDisco metamodel, which is an Ecore metamodel. The reverse engineering approach of MoDisco is very similar to the Ecore metamodel extraction approach of this thesis. Both the Ecore metamodel extraction implementation and MoDisco are based on the same Eclipse tools. However, there is one big difference between MoDisco and the extraction implementation. MoDisco extracts Ecore models, while the extraction implementation extracts Ecore metamodels. That means the approach of this thesis is on a higher metalevel (see Section 2.2). As a result, the extracted metamodels of the extraction implementation approach are on the same metalevel as the MoDisco metalevel, which describes all models extracted by MoDisco.

"MDA-Based Reverse Engineering of Object Oriented Code" [8] and *"Formalizing MDA-Based Reverse Engineering Processes"* [7] describe a reverse engineering approach that complies with Model Driven Architecture. They combine static and dynamic analysis in one process that generates MDA models. While this process internally uses EMOF metamodels, it extracts platform-specific UML models. While they use the same extraction base as the Ecore metamodel extraction, they reverse engineer a different type of model. UML models have some similarities with Ecore metamodels but serve a different purpose (see Chapter 2.3.1 and 2.6.1 in [31]).

"Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software" [2] proposes an approach for the reverse engineering of UML sequence diagrams from Java code through dynamic analysis. It defines its approach using metamodels and consistency rules. The generated UML sequence diagrams describe operational behavior. As a result, this approach is fundamentally different from the Ecore metamodel extraction.

"Shimba — an environment for reverse engineering Java software systems" [32] introduces the reverse engineering environment Shimba. Shimba extracts static software artifacts and their dependencies from Java Bytecode. With this information, SCED sequence

diagrams are extracted, which correspond to UML sequence diagrams. Additionally, Shimba allows reverse engineering a complete model of the dynamic behavior. This model can be represented in state charts. This work has two differences compared to the Ecore metamodel extraction. First, it uses Java Bytecode as extraction base, while the Ecore metamodel extraction uses Java source code. Second, it extracts behavioral UML models, which are not closely related with Ecore metamodels.

"Reverse Engineering of Object Oriented Code" [33] describes a unifying framework for reverse engineering code to various UML models such as object diagrams, class diagrams, interaction diagrams, state diagrams and package diagrams. These different UML models are extracted with the same static code analysis framework, which conducts the analysis of the code with a so-called Object Flow Graph. The Object Flow Graph is a representation of the program analyzed. This work explains methodology that is applicable on a broad basis. Some of the problems described in this work are similar to the problems encountered during the design of the Ecore metamodel extraction approach. The main difference between this work and the Ecore metamodel extraction is the object flow graph as core data structure compared to the *Intermediate Model*. While the *Intermediate Model* is designed for the extraction of Ecore metamodels alone, the object flow graph has to be able to serve for the extraction of many different UML models.

"Polymetric views - a lightweight visual approach to reverse engineering" [24] presents the concept of a polymetric view, which is a software visualization technique enriched with software metrics. It also describes a method which supports and guides software engineers in the initial phase of the reverse engineering process. Polymetric views are implemented in the tool CodeCrawler [23]. CodeCrawler relies on the FAMIXMetamodel [6], which allows to model languages such as C++, Java, Smalltalk, and COBOL. This approach is very different compared to the Ecore metamodel extraction in many ways. For example, it allows extracting from many different languages, including Java. More importantly, it serves more as a tool to guide software engineers in the early phases of a reverse engineering process of a large software system. As a result, this approach offers views that visualize software, which have significantly less functionality than metamodels.

There are also proprietary tools like Enterprise Architect [28] and Rational Software Architect Designer [27] that specifically support the reverse engineering of UML diagrams. These tools are widely used in the commercial sector.

At the current time, there are no existing approaches known to the author that allow the integration of Ecore functionality into Java code. However, there are approaches that, generally speaking, blur the line between models and code, which is in some ways similar to the goals of the Java code *Ecorification*.

One of such approaches is Xcore [38, 30]. It is an extended syntax for Ecore, which allows creating metamodels and their implementations textually. Meta information is added to the source code to describe the desired metamodel. During the compilation, the implementation of the metamodel is generated. This allows to remove the separation between metamodels and code and to combine the advantages of both models and code. In contrast to this thesis, it does not support the automatic integration of Ecore functionality into existing Java code.

"Projecting UML Class Diagrams from Java Code Models" [17] presents a prototype for a UML editor using the model representation of Java source code as a single underlying model. As a result, the code and the UML model are updated if one is changed. The UML editor is realized as a projection of a Java source code model and therefore does not need an explicit UML model. The goal of this work is to keep models and code consistent. This prototype tries to achieve consistency between Java code and UML models during the development process. The Java code *Ecorification* is a process that is meant to be used once for an existing program, with the goal to integrate the Ecore functionality. This is the main difference between this thesis and the prototype.

"Reverse Engineering of Object-Oriented Code into Umple using an Incremental and Rule-Based Approach" [10] presents a reverse engineering approach which adds modeling information incrementally to code written in C++ or Java while managing the system in a text format. This modeling information can be interpreted as a UML model. Because of its incremental process, this modeling information can be automatically changed and extended alongside the code. This approach is implemented in a tool called Umplificator. The Umple model is compatible with many UML and XMI formats. The concept behind the Umplificator is related to this thesis because of two reasons: First, it is a reverse engineering process from object-oriented code to UML models. Second, it gets rid of the distinction between model and code through adding the modeling information incrementally to code. However, its goal is entirely different, because in this thesis the reverse-engineering process is a means to an end.

9. Future Work

While the Ecore metamodel extraction is conceptually relatively complete and the prototype already implements many features, there are still some features that could extend the extraction. As previously mentioned, we currently extract fields of type `java.util.List` as *EStructuralFeature* which references the *EDataType* of `EList`. When manually creating such *EStructuralFeatures*, one usually uses multiplicities. That can be achieved through creating an *EStructuralFeature*, which has the type of the generic argument of the list, and then setting the upper bound of the multiplicity of the *EStructuralFeature* to minus one. This could be implemented for the Ecore metamodel extraction. Another prospective feature is the full extraction of enums. As explained in Section 4.2.2, an enum can only be partially extracted when representing it in the Ecore metamodel solely as an *EEnum*. To extract an enum with its methods, fields, constructors and realization relations, a single enum has to be extracted as one *EEnum* and one *EClass*. Other features for the Ecore metamodel extraction include the extraction of Javadoc [5, 19] comments through annotations and the extraction of the bodies of access methods. JavaDoc comments could be extracted with a particular kind of *EAnnotation*, which has the type value `EModelElement` and key value `documentation`. The bodies of access methods could be extracted through custom *EAnnotations* to retain the access methods with additional functionality.

Another idea is to use the Ecore metamodel extraction to extract an Ecore metamodel of the *Intermediate Model*. The extracted metamodel could then replace the *Intermediate Model* in the Ecore metamodel extraction. This bootstrapping would allow for example model-to-model transformations from and to the *Intermediate Model* with transformation languages like QVT [13] or ATL [16]. In the future, the *Intermediate Model* could be adapted to fit any object-oriented programming language. As a result, it would be possible to extract Ecore metamodels from other programming languages through generating an *Intermediate Model* from source code of these languages.

The concept for the *Ecorification* is still just a rough prototype and its prototypical implementation just a proof of concept. For this reason, there is much room for improvements and new features for the Java code *Ecorification*. Prospective features for the implementation are the removal of the fields from the *adapted origin code* (see Section 5.5), the adaption of the Ecore factories (see Section 5.6) and a solution for the constructor problem (see Section 5.7). During the validation, we discovered the problem with the contravariance of the return types of parameterized accessor methods, which occurs when extracting a field of type `java.util.List` during the *Ecorification*. This has to be solved in the near future.

10. Conclusion

Many tools rely on the patterns of code generated from Ecore metamodels. To enable the use of these tools on existing Java code, this thesis introduced an approach for the automatic integration of Ecore functionality into arbitrary Java code. We called this approach *Ecorification* of Java code. During the integration of Ecore functionality, the Java code *Ecorification* preserves the original functionality of the Java code and all its interfaces. To make the *Ecorification* possible, this thesis also introduced a reverse engineering approach, which extracts Ecore metamodels from Java code. This reverse engineering approach is called Ecore metamodel extraction.

The *Ecorification* of Java code first creates an Ecore representation of the original Java code, called *origin code*. The Ecore representation consists out an Ecore metamodel that represents the Java code as closely as possible and the model code generated from that Ecore metamodel, which we call *Ecore code*. The metamodel is created with the Ecore metamodel extraction, which extracts the metamodel from the *origin code*. At the core of the Ecore metamodel extraction is a mapping from elements of the implicit Java metamodel to elements of the Ecore meta-metamodel. This mapping defines how Ecore metamodel elements are extracted from Java code. The similarities of the *origin code* and the *Ecore code* are then used to interlace them both. This is achieved by utilizing the separation of interface and implementation of the model code to mount the *origin code* into the super relation hierarchy of the *Ecore code*. For that, the *Ecorification* uses wrapper classes. They form the *integration code*. The *origin code* is adapted to allow extending the wrapper classes of the *integration code*. The wrapper classes then are mounted into the hierarchy of the model code. Finally, all three codes are combined to the *decorated code*. The *decorated code* contains the implementation details of the *origin code* and the Ecore functionality of the *Ecore code*.

The concept of this thesis was implemented in two parts: First, the Ecore metamodel extraction was implemented. Second, the extraction implementation was used to implement a prototype of the Java code *Ecorification*. While the extraction implementation covers almost all features of the concept, the *Ecorification* prototype is only implemented as a proof of concept. Both the *Ecorification* and the extraction were implemented as Eclipse plug-ins in less than 3200 lines of code combined, using the Java Development Tools (see Section 2.5) and the Ecore API (see Chapter 14.3 in [31]). While the Ecore metamodel extraction approach was primarily developed for the Java code *Ecorification*, it can also be independently used as a reverse engineering tool.

Initially, the Ecore metamodel extraction was separately validated. After that, the whole *Ecorification* was validated, which implicitly confirmed the validity of the Ecore metamodel extraction again because it is a part of the *Ecorification*. To validate the Ecore metamodel extraction, the extracted Ecore metamodels were examined on validity using three indicators: The validity of the extracted metamodel according to EMF, the ability to generate

Ecore model code from the extracted metamodel, and indication through examined model properties. The examined model properties were compared to the correlating properties of the Java code used for the extraction. We tested the extraction for a custom designed test project and two larger projects: The Java AST, a subset of the Java Development Tools (see Section 2.5) and the Apache Commons IO library [26]. The Java code *Ecorification* was validated through performing a QVTO transformation [13] on Java code, into which Ecore functionality was integrated through the *Ecorification* of Java code. Because QVTO transformations cannot be conducted on regular Java code, this validation proved that we were able to successfully integrate Ecore functionality into Java code.

The successful validation of this thesis indicates that the *Ecorification* implementation integrated enough Ecore functionality into existing Java projects in order to allow the application of an Ecore-dependent tool on arbitrary Java code. As a result, we conclude that the Java code *Ecorification* accomplishes its goal. Hence, this thesis proves the feasibility of the automatic integration of Ecore functionality into Java code.

Bibliography

[1] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. 4th ed. The Java series. Upper Saddle River, NJ: Addison-Wesley, 2006. ISBN: 0-321-34980-6. DOI: 10.1002/9780470693698.ch1. URL: http://etf.beastweb.org/index.php/site/download/Java_Programming.pdf.

[2] Lionel C Briand, Yvan Labiche, and Johanne Leduc. "Toward the Reverse Engineering of UML Sequence Diagrams for Distributed Java Software". In: *IEEE Transactions on Software Engineering* 32.9 (Sept. 2006), pp. 642–663. DOI: 10.1109/tse.2006.96. URL: https://doi.org/10.1109%2Ftse.2006.96.

[3] Hugo Bruneliere et al. "MoDisco: A Generic And Extensible Framework For Model Driven Reverse Engineering". In: *Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE '10*. ACM. Association for Computing Machinery (ACM), 2010, pp. 173–174. DOI: 10.1145/1858996.1859032. URL: https://doi.org/10.1145%2F1858996.1859032.

[4] Gerardo CanforaHarman and Massimiliano Di Penta. "New Frontiers of Reverse Engineering". In: *2007 Future of Software Engineering*. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 326–341. ISBN: 0-7695-2829-5. DOI: 10.1109/FOSE.2007.15. URL: http://dx.doi.org/10.1109/FOSE.2007.15.

[5] Oracle Corporation. *Javdoc*. Oracle.com. Version 1.5.0. Feb. 2004. URL: http://www.oracle.com/technetwork/java/javase/documentation/javadoc-137458.html (visited on 04/09/2017).

[6] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. *FAMIX 2.1—the FAMOOS information exchange model*. 2001.

[7] Liliana Favre. "Formalizing MDA-Based Reverse Engineering Processes". In: *2008 Sixth International Conference on Software Engineering Research, Management and Applications*. Institute of Electrical and Electronics Engineers (IEEE), 2008. DOI: 10.1109/SERA.2008.21. URL: https://doi.org/10.1109%2Fsera.2008.21.

[8] Liliana Favre, Liliana Martinez, and Claudia Pereira. "MDA-Based Reverse Engineering of Object Oriented Code". In: *Enterprise, Business-Process and Information Systems Modeling: 10th International Workshop, BPMDS 2009, and 14th International Conference, EMMSAD 2009, held at CAiSE 2009, Amsterdam, The Netherlands, June 8-9, 2009. Proceedings*. Ed. by Terry Halpin et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 251–263. ISBN: 978-3-642-01862-6. DOI: 10.1007/978-3-642-01862-6_21. URL: http://dx.doi.org/10.1007/978-3-642-01862-6_21.

[9] Eclipse Foundation. *Eclipse Public License*. Eclipse.org. Version 1.0. Feb. 2004. URL: https://www.eclipse.org/legal/epl-v10.html (visited on 04/04/2017).

[10] Miguel A. Garzón et al. “Reverse Engineering of Object-oriented Code into Umple Using an Incremental and Rule-based Approach”. In: *Proceedings of 24th Annual International Conference on Computer Science and Software Engineering*. CASCON ’14. Markham, Ontario, Canada: IBM Corp., 2014, pp. 91–105. URL: `http://dl.acm.org/citation.cfm?id=2735522.2735534`.

[11] James Gosling and Henry McGilton. “The Java Language Environment”. In: *Sun Microsystems Computer Company* 2550 (1995). URL: `http://www.oracle.com/technetwork/java/langenv-140151.html`.

[12] James Gosling et al. *The Java Language Specification*. Vol. 8. Addison-Wesley Professional, 2015. URL: `http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf`.

[13] Object Management Group. “Meta Objet Facility (MOF) 2.0 Query/View/Transformation”. Version 1.3. In: (June 2016). URL: `http://www.omg.org/spec/QVT/1.3/`.

[14] Graham Hamilton. “Java Beans Specification”. In: *Sun Microsystems* 1.01 (1997). URL: `http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html`.

[15] Ayushman Jain and Stephan Herrmann. *How to Train the JDT Dragon*. EclipseCon Talk. Mar. 2012. URL: `http://www.eclipsecon.org/2012/sites/eclipsecon.org.2012/files/How%20To%20Train%20the%20JDT%20Dragon%20combined.pdf` (visited on 04/04/2017).

[16] Frédéric Jouault et al. “ATL: A QVT-like Transformation Language”. In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA ’06. Portland, Oregon, USA: ACM, 2006, pp. 719–720. ISBN: 1-59593-491-X. DOI: `10.1145/1176617.1176691`. URL: `http://doi.acm.org/10.1145/1176617.1176691`.

[17] Heiko Klare, Michael Langhammer, and Max E. Kramer. “Projecting UML Class Diagrams from Java Code Models”. In: *4th Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling : Proceedings, 2 March 2016, Karlsruhe, Germany. Ed.: C. Atkinson.* Institut für Programmstrukturen und Datenorganisation (IPD), 2016, pp. 11–18. URL: `https://sdqweb.ipd.kit.edu/publications/pdfs/klare2016a.pdf`.

[18] Ralf Kollmann et al. “A study on the current state of the art in tool-supported UML-based static reverse engineering”. In: *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* Institute of Electrical and Electronics Engineers (IEEE), 2002, pp. 22–32. DOI: `10.1109/wcre.2002.1173061`. URL: `http://dx.doi.org/10.1109/WCRE.2002.1173061`.

[19] Douglas Kramer. “API Documentation from Source Code Comments: A Case Study of Javadoc”. In: *Proceedings of the 17th Annual International Conference on Computer Documentation*. SIGDOC ’99. New Orleans, Louisiana, USA: ACM, 1999, pp. 147–153. ISBN: 1-58113-072-4. DOI: `10.1145/318372.318577`. URL: `http://doi.acm.org/10.1145/318372.318577`.

[20] Max E. Kramer, Erik Burger, and Michael Langhammer. “View-centric engineering with synchronized heterogeneous models”. In: *Proceedings of the 1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling*. VAO ’13. Montpellier, France: ACM, 2013, 5:1–5:6. ISBN: 978-1-4503-2070-2. DOI: `10.1145/2489861.2489864`. URL: `http://doi.acm.org/10.1145/2489861.2489864`.

[21] Max E. Kramer et al. “Change-Driven Consistency for Component Code, Architectural Models, and Contracts”. In: *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering*. CBSE ’15. Montréal, QC, Canada: ACM, 2015, pp. 21–26. ISBN: 978-1-4503-3471-6. DOI: `10.1145/2737166.2737177`. URL: `http://doi.acm.org/10.1145/2737166.2737177`.

[22] Thomas Kuhn and Oliver Thomann. “Abstract Syntax Tree”. In: *Eclipse Corner Articles* 20 (Nov. 2006). URL: `http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html`.

[23] Michele Lanza. “CodeCrawler-polymetric views in action”. In: *Proceedings of the 19th IEEE international conference on Automated software engineering*. IEEE Computer Society. Institute of Electrical and Electronics Engineers (IEEE), 2004, pp. 394–395. DOI: `10.1109/ase.2004.1342773`. URL: `https://doi.org/10.1109%2Fase.2004.1342773`.

[24] Michele Lanza and Stéphane Ducasse. “Polymetric views - A lightweight visual approach to reverse engineering”. In: *IEEE Transactions on Software Engineering* 29.9 (Sept. 2003), pp. 782–795. DOI: `10.1109/TSE.2003.1232284`. URL: `http://dx.doi.org/10.1109/TSE.2003.1232284`.

[25] Ed Merks. *Eclipse Forum Entry*. www.eclipse.org/forums/index.php/t/209028. July 2009. (Visited on 03/14/2017).

[26] Apache Software Foundation. *Commons IO*. Apache.org. Version 2.5. Apr. 2016. URL: `https://commons.apache.org/proper/commons-io/` (visited on 04/05/2017).

[27] Rational Software. *Rational Software Architect Designer*. IBM.com. Version 9.5. Sept. 2015. URL: `https://www-03.ibm.com/software/products/de/ratsadesigner` (visited on 04/07/2017).

[28] SparxSystems. *Enterprise Architect*. sparxsystems.com. Version 13 Build 1309. Nov. 2016. URL: `http://www.sparxsystems.com/products/ea/index.html` (visited on 04/07/2017).

[29] Thomas Stahl and Markus Völter. *Model Driven Software Development: Technology, Engineering, Management*. Chichester: Wiley, 2006. ISBN: 0-470-02570-0; 978-0-470-02570-3.

[30] Alexandru Ştefănică and Petru Florin Mihancea. “XCORE: Support for developing program analysis tools”. In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. Institute of Electrical and Electronics Engineers (IEEE), Feb. 2017, pp. 462–466. DOI: `10.1109/saner.2017.7884654`. URL: `https://doi.org/10.1109%2Fsaner.2017.7884654`.

[31] Dave Steinberg, ed. *EMF - Eclipse modeling framework.* 2nd ed. The eclipse series. Boston, Mass.: Addison-Wesley, 2009. ISBN: 0-321-33188-5; 978-0-321-33188-5.

[32] Tarja Systä, Kai Koskimies, and Hausi Müller. "Shimba — an environment for reverse engineering Java software systems". In: *Software: Practice and Experience* 31.4 (2001), pp. 371–394.

[33] Paolo Tonella. *Reverse Engineering of Object Oriented Code.* Springer New York, 2005. DOI: 10.1007/b102522. URL: http://dx.doi.org/10.1007/b102522.

[34] Paolo Tonella and Alessandra Potrich. "Reverse engineering of the UML class diagram from C++ code in presence of weakly typed containers". In: *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001.* Institute of Electrical and Electronics Engineers (IEEE), 2001, pp. 376–385. DOI: 10.1109/icsm.2001.972750. URL: http://dx.doi.org/10.1109/ICSM.2001.972750.

[35] Antonio Vallecillo et al. "Formal Specification and Testing of Model Transformations". In: *Formal Methods for Model-Driven Engineering.* Springer Nature, 2012, pp. 399–437. DOI: 10.1007/978-3-642-30982-3_11. URL: https://doi.org/10.1007%2F978-3-642-30982-3_11.

[36] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. "Sirius: A rapid development of DSM graphical editor". In: *Intelligent Engineering Systems (INES), 2014 18th International Conference on.* IEEE. 2014, pp. 233–238. URL: http://ieeexplore.ieee.org/abstract/document/6909375/.

[37] Lars Vogel. *Eclipse Modeling Framework (EMF) - Tutorial.* vogella.com. Version 3.6. July 2016. URL: http://www.vogella.com/tutorials/EclipseEMF/article.html (visited on 04/04/2017).

[38] Sabine Winetzhammer and Bernhard Westfechtel. "ModGraph meets Xcore: Combining rule-based and procedural behavioral modeling for EMF". In: *Electronic Communications of the EASST* 58 (2013). URL: http://dx.doi.org/10.14279/tuj.eceasst.58.838.

A. Appendix

A.1. Predefined EDataTypes

Java Type	Ecore Type
boolean	EBoolean
byte	EByte
char	EChar
double	EDouble
float	EFloat
int	EInt
long	ELong
short	EShort
java.lang.Boolean	EBooleanObject
java.lang.Byte	EByteObject
java.lang.Character	ECharacterObject
java.lang.Double	EDoubleObject
java.lang.Float	EFloatObject
java.lang.Integer	EIntegerObject
java.lang.Long	ELongObject
java.lang.Short	EShortObject
java.lang.String	EString
java.lang.Object	EJavaObject
java.lang.Class	EJavaClass

Table A.1.: Predefined *EDataTypes* and their Java counterparts (see Chapter 5.8 in [31]).

A.2. Ecore Meta-Metamodel Hierarchy

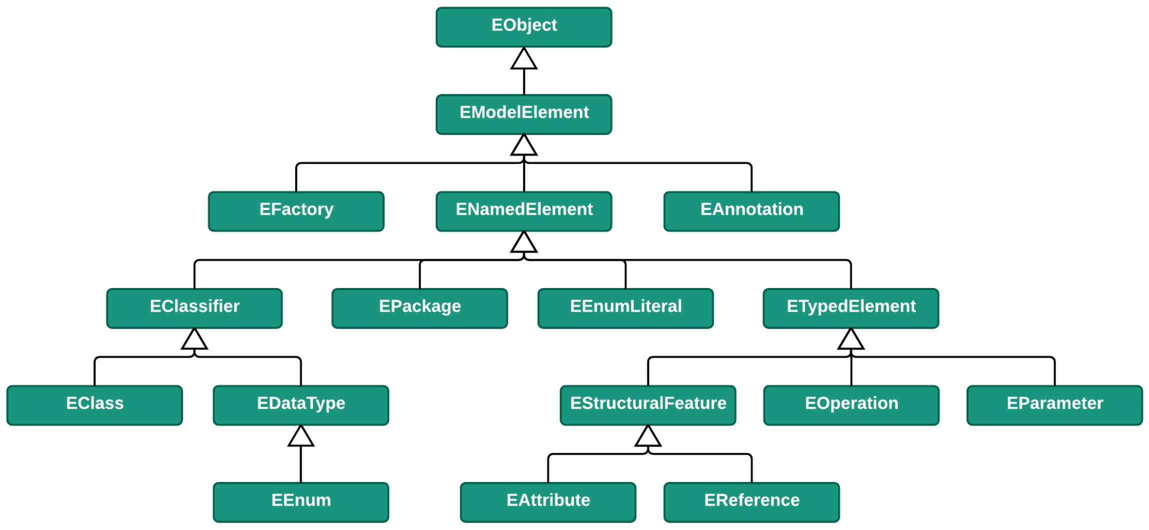


Figure A.2.: UML Class Diagram of the Ecore meta-metamodel hierarchy. It shows the relations of the Ecore meta-metamodel elements (see Chapter 5 in [31]).