Software and Systems Modeling (2022) 21:1689–1709
https://doi.org/10.1007/s10270-022-01034-9

EXPERT VOICE

An executable metamodel refactoring catalog

Lorenzo Bettini[1] · Davide Di Ruscio[2] · Ludovico Iovino[3] · Alfonso Pierantonio[2]

Received: 13 June 2022 / Accepted: 18 July 2022 / Published online: 12 August 2022

Abstract
Like any software artifacts, metamodels are evolving entities that constantly change over time for different reasons. Changing metamodels by keeping them consistent with other existing artifacts is an error-prone and tedious activity without the availability of automated support. In this paper, we foster the adoption of metamodel refactorings collected in a curated catalog. The Edelta framework is proposed as an operative environment to provide modelers with constructs for specifying basic refactorings and evolution operators, to define a complete metamodel refactoring catalog. The proposed environment has been used to implement the metamodel refactorings available in the literature and make them executable. A detailed discussion on how modelers can use and contribute to the definition of the catalog is also given.

Keywords Metamodels · Evolution · Refactoring · Catalog

1 Introduction

Refactoring is "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" [1]. Refactoring, in general, can be defined as a disciplined way to clean up code with the intent of minimizing the chances of introducing bugs [2]. "Improving the design after it has been written" is the definition Fowler gives to a code refactoring [3]. In fact, refactoring is the opposite of the good practice where we design and then produce the code. With refactoring, bad design can be improved and transformed into well-designed code.

Model-Driven Engineering (MDE) shifts the focus from coding to modeling. Models are artifacts defined at a higher level of abstraction, offering advantages in terms of productivity and quality over the entire development life-cycle. Domain-Specific Modeling Languages (DSMLs) formalize the system structure, behavior, and requirements within particular application domains. DSMLs are described using *metamodels*, which define the relationships among concepts and constraints. Like other software artifacts, metamodels are living entities that constantly change and evolve over time [4,5] for different reasons [6], as, for instance, when new insights about the domain emerge and require to be consistently reflected in the metamodel formalization, or for improving the quality of it. Indeed, as for code bad structure, metamodels can also be designed with issues discovered in a future development stage. Therefore, whenever a metamodel changes, all the corresponding entities must be accordingly adapted in order to remain valid [7]. The modeler might adapt the corrupted artifacts by inspecting them, detecting the needed refactorings according to the applied corresponding metamodel changes, and applying them manually. However, this process is based on individual skills and specific cases of complexity of the artifact. Moreover, it is an error-prone activity, resulting tedious without automated support [8].

One of the activities enabling the automated management of metamodel evolutions is their specification [9]. The work in [10] is the first attempt at classifying metamodel changes, grouping them into sub-groups of renamings, deletions, additions, movements, association changes, and type changes. However, during the last years, we discovered that metamodel changes could be classified in a more structured way as code refactorings. The need for a formalized metamodel refactorings catalog is highlighted by the mentions

Communicated by Bernhard Rumpe.

Fully documented templates are available in the elsarticle package on CTAN.

✉ Ludovico Iovino
ludovico.iovino@gssi.it

[1] University of Florence, Florence, Italy

[2] University of L'Aquila, L'Aquila, Italy

[3] Gran Sasso Science Institute, L'Aquila, Italy

in the literature of metamodel changes, also occurring in real case scenarios, without any form of standardization, as Fowler instead does [3]. Moreover, metamodel refactoring operations can be simple or complex, by aggregating and composing them.

In this paper, we use the term "refactoring" by slightly abusing the definition to include semantic-preserving metamodel changes, even if, in some cases, this may depend on the domain-specific concepts we are evolving. For instance, merging two metaclass properties may preserve or not the semantic meaning of those properties or can be the result of an intentional change applied to the metamodel, as structural changes. Many papers in the literature implicitly or explicitly refer to metamodel refactoring operations, in some cases also by naming them. These works tend to use different representations for metamodel changes, from state-based to operation-based and different technical spaces. Analyzing the literature, we discovered that most of these works define their catalog of changes, some of them referring to other approaches.

In September 2011, we began to collect metamodel refactoring patterns on a website, available at www.metamodelrefactoring.org. This website collects refactoring patterns, the formal definition, and the diagrams of an exemplary metamodel before and after the subject refactoring. This website was intended for internal and student use, but it demonstrated that having an organized catalog would ease the work of relating to an existing pattern of refactoring for metamodels in a more structured and standardized way. Over the last few years, many works cited the metamodel refactoring catalog website, e.g., [11–27]. Even if the website is not a scientific publication, it received a lot of attention. This motivated us particularly about the need of improving the way the catalog is defined, organized, and how the community can contribute with adding new content.

This paper presents our recipe to build a metamodel refactoring catalog. We use Edelta [21], a framework that provides modelers with constructs for specifying basic refactorings and evolution operations, to define a complete metamodel refactoring catalog. We have implemented the catalog formally presented in [28] making it executable with a dedicated framework. We also show how a modeler can use the catalog, contribute with her definitions, and productively maintain the catalog.

Structure of the paper Section 2 introduces the background and motivates the need for building a catalog of metamodel refactorings. In Sect. 3, we show our recipe for building a catalog of metamodel refactorings, which we show in detail in Sect. 4. Section 5 describes how it is possible to contribute to the catalogue by relying on the proposed technology. Finally, we draw the conclusions and future works in Sect. 6.

2 Background and motivation

Refactoring is a well-known technique for restructuring an existing body of code, changing properly its internal structure without changing its external behavior [3]. In modeling, refactoring can be seen as "a perfective evolution of models and in particular domain models" [29]. In metamodeling languages, refactoring operations can be operated for different purposes, e.g., for introducing new requirements or refining existing ones. By exploring the topic of metamodel refactoring in the literature, we noticed that metamodels are subject to many changes during their life-cycle [30]. We have conducted a literature review on the topic with the aim of answering the following research questions:

- **RQ1**: *What is the relevance of the metamodel refactoring topic in the modeling community?*
- **RQ2**: *What are the intents of metamodel refactorings?*

To this end, we defined the following search string used to query Google Scholar:

> "Metamodel and (intitle:refactoring or evolution) and MDE"

The query returned 297 results, subsequently filtered employing inclusion and exclusion criteria. In particular, the inclusion criteria we applied are:

I1: The work is included if at least a concrete metamodel refactoring/evolution operator is defined in the paper under analysis;
I2: The work defines a metamodel refactoring with a "name" identifying the operator's goal, e.g., merge attributes, rename class.

The considered exclusion criteria are:

E1: The paper is excluded if the refactoring is not applied at the metamodeling level;
E2: The paper is excluded if an example of application is not reported in the paper;
E3: The paper is excluded if it is not written in English;

Table 1 shows the papers that have been obtained after the application of the inclusion and exclusion criteria on the initial 297 works retrieved by querying Google Scholar. The table characterizes the obtained 16 papers concerning: (i) the way metamodel change operators are specified (i.e., operation-based, state-based, or implicitly described in the text); (ii) the intent of the discussed metamodel change operators, i.e., for improving the quality of the analyzed metamodels, for supporting coupled evolution scenarios, to

Table 1 Collected works

	Specification	Intent	Reuse method	Catalog size
[28]	–	Co-evolution	–	60
[22]	Operation-based	Quality enhancement	Repository	5
[32]	State-based	Co-evolution	–	10
[33]	State-based	Co-evolution	–	10
[34]	State-based	Co-evolution	–	7
[35]	State-based	Co-evolution	–	10
[36]	Operation-based	Detection	–	26
[37]	State-based	Co-evolution	Transformation rules	3
[38]	State-based	Co-evolution	-	15
[39]	Operation-based	Refactoring	Transformation rules	5
[40]	Implicit	Co-evolution	–	1
[41]	State-based	Co-evolution	–	3
[42]	Operation-based	Co-evolution	–	2
[43]	State-based	Co-evolution	–	17
[44]	Operation-based	Refactoring	Documentation	17
[45]	State-based	Quality enhancement	Repository	5

detect metamodel evolution, or in general to refactor the metamodels at hand; (iii) the provided means (if any) to enable the reuse of available metamodel change operators; (iv) the size of the presented catalog of metamodel change operators.

Most of the analyzed works treating metamodel refactorings are for quality enhancement or coupled evolution. This is highlighted from the column "Intent" of Table 1, where we have classified the papers into three categories, i.e., coupled evolution, quality enhancement, and refactoring. The first category proposes approaches for restoring the corrupted relation between the evolving metamodel and artifacts defined on top of it. The second category is improving the quality of the subject metamodel by using refactoring techniques, e.g., to solve emerging bad smells. Finally, the third category includes approaches to deal with the evolution/refactoring of the subject metamodel.

Concerning the specification of metamodel refactorings, we identified three kinds of approaches, i.e., state-based, operation-based, and implicit. State-based approaches store the states of a given metamodel and thus need to derive the changes by comparing two "states," i.e., two versions of the same metamodel. Operation-based approaches are a particular class of change-based approaches representing the changes as transformation operations [31]. These operations are applied to the initial metamodel to get the refactored version. Finally, implicit representations are not concretized in any technical space but computed following an algorithm specification at runtime.

Interestingly, in [28] the authors present a catalog of operators targeting as intent the coupled evolution of models/metamodels problem. It is the first work giving a catalog of reusable coupled operators covering metamodel evolution and model migration. This presented catalog consists of nearly 60 operators, which are properly detailed even though no supporting implementation is available. The notation used to describe the refactorings is textual and needs to be followed to implement it in specific technical spaces. However, the reuse method is unavailable since it is a well-documented catalog without any implemented tool for application, execution, and reuse of refactoring operators.

Bettini et al. [22] uses the *Edelta* language for specifying metamodel evolution patterns to identify bad smells and solve them through refactoring operators. The paper proposes a catalog of defined refactoring using the presented DSL, and the intent was the quality improvement of metamodels as [45].

The paper in [32] supports a state-based representation of refactoring operators that are specified through formal specification rules defining co-evolution scenarios. However, there is no specific reuse mechanism since the approach does not seem to be implemented.

Di Ruscio et al. [33,35] express metamodel refactorings via a state-based representation, which is automatically calculated by comparing two versions of the same metamodel. In particular, the intent is to use the proposed state-based representation to guide the coupled evolution of corrupted GMF models [33], and the definition of textual concrete syntaxes [35]. Since the metamodel changes are calculated based on a defined algorithm for model differencing, there is no specific collection method. For this reason, reuse is not supported since the refactoring specification is domain-specific and not metamodel-independent.

Basciani et al. [34] use a state-based representation of the metamodel changes to guide the co-evolution of multiple types of artifacts.

The approach presented in [36] reconstructs complex metamodel changes based on a algorithm-encoded patterns on the two versions of metamodels. The intent of this work is then detection resolution of metamodel changes, and it is based on trace analysis and then, classified operation-based.

Garces et al. [37] use weaving models for co-evolving transformations w.r.t. metamodel changes. Also, in this case, refactorings applied to metamodels are represented with a state-based approach, and reuse is partially supported since transformation rules are used to apply metamodel refactorings, which can be reused and adapted by other users. Another work with the same intent, i.e., co-evolution, is reported in [38], where a state-based representation is computed based on the similarity between two metamodels.

Quality is the intent of the work in [45], where the authors use Alloy to specify metamodel refactorings and bad smell finders so that the metamodel quality can be automatically improved. Reuse is supported since the Alloy specifications are shared through a public GitHub repository.

The work in [39] uses an operation-based approach to represent refactorings via visual graph transformation rules. Also, in this case, since rules can be shared with other users, the need for a shared catalog of refactorings is highlighted.

Rose et al. [40] used an implicit representation for metamodel changes to support corresponding model adaptations. However, reuse is unsupported since refactoring operations are not directly explorable by users and are embedded in a hash differencing algorithm.

A dedicated metamodel for representing metamodel changes as state-based is proposed in [41]. The approach is used to co-evolve code generators w.r.t. metamodel changes.

In [42], a state-based approach is proposed for representing metamodel changes through templates tackling the evolution of defined constraints.

Visual examples representing metamodel evolutions and adaptations in a multi-level modeling setting [46] are reported in [43]. Reuse is missing because the multi-level modeling setting still does not have mature tools for implementing the proposed approach.

In [44], the authors define metamodel changes by means of pseudo-code examples for refactoring metamodels. Reuse is not supported since the approach is not implemented.

By analyzing the papers previously discussed, we identified the following requirements, which should be satisfied when building and managing a catalog of metamodel refactorings.

$R1$. **Defined by a domain-specific language** The first identified requirement is that a dedicated language is needed to define refactoring operations that can be shared and reused by the community. This is a fundamental requirement since refactoring operators may be atomic or complex ones. Complex refactorings can be defined in terms of other atomic operations to reuse them or to give more intentional definitions. Moreover, using a dedicated language delivers several advantages, including *executability* and the fact that domain-specific languages (DSLs) are natural enablers for reuse [47]. Moreover, using DSLs reduces the need for learning some of the specialized metalanguages, e.g., Alloy used in [45], and it results in being more intuitive for domain experts. Refactoring operations should be defined so that they are reusable in the same refactoring program or by sharing the definition with other users. However, re-defining the same refactoring operation multiple times can affect the quality of the refactoring process and is also time-consuming. Moreover, if we imagine the process as a transformational approach of the input metamodel, not defining reusable operations would mean re-compile the same procedure multiple times, affecting the performance of the entire transformation.

$R2$. **Supporting qualitative process** From the previous requirement, a defined operation should assure a qualitative result, as confirmed by the analyzed works with having the intent of improving the quality of metamodels. For this reason, a quality-driven process for the definition is needed to assure that the defined refactorings produce improved or same-quality metamodels, but not worse. A refactoring operation should be available for testing the effect on a set of metamodels before confirming the definition or sharing it with other users. A wrongly defined refactoring could create a ripple effect on the quality of a metamodel. Indeed, not always a refactoring produces good quality results [48,49]. Therefore, the definition of the catalog should be supported by continuously testing the defined operators.

$R3$. **Mitigating the generation of corrupted models** When refactorings are defined and applied, the subject metamodel can be inconsistent. Indeed, if the refactoring acts on the metamodel with wrongly described operations, the result can be a corrupted model. Existing works deal with erroneous metamodel with an activity which is called *model repair* [50], which of course should be avoided in advance.

$R4$. **Open to contribution** Collecting refactoring operations in a repository should be open to external contributions, refining, or even building on existing refactorings. For instance, an extract metaclass refactoring can be reused to define an extract superclass. This requirement also contributes to the execution performance previously defined.

$R5$. **Executable** A defined refactoring operation should be executable. Thus, a refactoring operator should define the executable semantic [9] of the refactoring definition.

$R6$. **Available in open repositories** Standardization is a goal that should be targeted in this domain. For this reason, a repository of refactoring definitions for metamodels may be beneficial. To the best of our knowledge, only [28] defined a complete catalog of 60 operators, but they are only textually defined without the availability of a corresponding execution engine.

Table 2 Ingredients of the proposed approach implementing the identified requirements

	*R*1	*R*2	*R*3	*R*4	*R*5	*R*6
Edelta DSL (Sect. 3.1)	•		•			
Intepreter, Compiler, and IDE (Sect. 3.2)		•	•		•	
Runtime Library (Sect. 3.3)		•			•	
Refactoring Catalog (Sect. 4)				•		•
Automated tests (Sect. 5)		•	•		•	

3 Our recipe for building a metamodel refactorings catalog

In this section, we present a technical solution satisfying the requirements in Sect. 2. In particular, as shown in Table 2, we propose a metamodel refactoring framework consisting of a DSL for defining refactorings and implementing refactoring programs (see Sects. 3.1–3.3). By using the proposed framework, we built a catalog of refactorings made available to the community with a dedicated refactoring library that modelers can import and reuse in their programs (Sect. 4). We also propose a collaborative process to contribute to the catalog and test the defined operators (Sect. 5).

3.1 Edelta: a framework for metamodel refactoring

This section describes the Edelta framework for specifying and applying metamodel refactorings. Edelta was first presented in [21], and an updated version was subsequently presented in [51]. In the following, we describe in more detail the main features of Edelta that contribute to the implementation of the requirements listed in Sect. 2. Interested readers can refer to [51] for a detailed description of all the Edelta features.

Edelta can be installed directly using the Eclipse update site or downloading a complete Eclipse distribution with Edelta installed. Moreover, a Maven plugin to compile Edelta programs in a headless way is available, e.g., in a Continuous Integration server.[1]

The overall implementation of Edelta consists of two collaborating parts: (i) a Java runtime library, which implements the basic operations for manipulating metamodels, expressed as Ecore models (Sect. 3.3); (ii) a DSL that is compiled into Java code that uses the Edelta runtime library, executing the refactoring program. The Edelta DSL focuses on many static checks to catch most problems during the compilation of the program. The generated Java code, when executed, does not create an invalid metamodel. The Edelta DSL has been implemented with Xtext [52], the most popular Eclipse framework for the development of programming languages and DSLs. The Edelta framework offers a live view on the evolution of the metamodel while the user is writing an Edelta program in the Eclipse Edelta editor (Sect. 3.2). From now on, we refer to the Edelta DSL simply by Edelta unless we want to stress a particular aspect of the overall Edelta implementation.

Given an Edelta program, the Edelta compiler generates a Java file, which relies on the Edelta Java runtime library (see Sect. 3.3). Note that the generated Java code has no further dependency on the Edelta compiler or the Xtext framework itself: it depends only on the Edelta runtime library, which weights only a few megabytes. This means that the generated Java code that executes the actual evolution of Ecore files can be executed independently from Eclipse and the whole Xtext infrastructure.

In Edelta programs, the manipulated Ecore file, i.e., the subject metamodel, must be explicitly specified by the name, using the instruction metamodel "...". These include both the EPackages under modifications and the EPackages referred to in the program. Content assist is provided by the Edelta editor both for Java-like imports and for importing the Ecore files that are available in the current project's classpath.

An Edelta program can then contain definitions of "reusable operations," meant to specify common general refactorings, which can be reused both by the same program and by other Edelta programs with the clause use ...as .... A reusable operation is defined in Edelta with the following syntax:

```
1 def <name>(<... parameters ...>) : <returntype> {
2     <body>
3 }
```

Parameter declarations have the same syntax as in Java. The return type can be omitted and will be inferred by the compiler (making the return type explicit can help documenting the operation).

Moreover, an Edelta program can contain modification operations for specific metamodels, which are not meant to be reused outside that Edelta program. A modification operation for an imported EPackage is defined with the following syntax:

```
1 modifyEcore <name> epackage <EPackage name> {
2     <body>
3 }
```

Here, <name> is a unique name to identify this modification operation and <EPackage name> is the name of an EPackage that must be present in one of the imported metamodels (see the instruction metamodel "..." described above).

[1] The Edelta open-source project is available at https://github.com/LorenzoBettini/edelta.

Edelta uses Xbase [53] to provide a rich Java-like syntax for its expressions. Indeed, the <body> part consists of Xbase expressions. In order to make code snippets presented in the paper comprehensible, in the next section we describe the main features of Xbase that are used by Edelta. For further details, we refer to [21].

Xbase is an extensible and reusable expression language that is completely interoperable with Java and its type system. Xbase is meant to be embedded in an Xtext DSL for implementing expressions (syntax, semantics and code generation into Java). The syntax of Xbase is Java-like; thus, it should be easily understood by Java programmers. However, Xbase aims at avoiding much of the "syntactic noise" verbosity that is typical of Java, so that it can be easily used by modelers.[2]

The complete interoperability with Java implies that Edelta seamlessly reuses the Java type system, including generics. All existing Java types can be referred and used from within an Xbase expression, that is, all existing Java libraries can be seamlessly reused in Edelta. From a typing point of view, Xbase expressions have the short and readable shape that might be typical of a dynamically (i.e., untyped) language, while retaining all the benefits of a statically typed language. Types can be omitted in many places, but of course the compiler will statically check that expressions are well-typed.

Variable declarations in Xbase start with val or var, for final and non-final variables, respectively. The special syntax for collections, #[e1, …, en], allows the programmer to easily specify a list with initial contents. Moreover, Xbase provides syntactic sugar for getters and setters: instead of writing o.getName(), one can simply write o.name; similarly, instead of writing o.setName("..."), one can write o.name = "...".

Another useful feature is *extension methods*, which are a syntactic sugar mechanism to simulate adding new methods to existing types without modifying them. Instead of passing the first argument inside the parentheses of a method invocation, the method can be called with the first argument as its receiver, as if the method was one of the argument type's members. For example, if m(Entity) is an extension method, and e is of type Entity, you can write e.m() instead of m(e), as if m was a method defined in Entity.

In Xbase, *lambda expressions* have the shape: [param1, param2, ... | body]. The types of parameters can be omitted if they can be inferred from the context. When a lambda is the last argument of a method call, it can be moved out of the parenthesis: instead of writing m(..., [...]), one can write m(...)[...].

Besides this, Xbase has another additional special variable, it. Similarly to this, it can be omitted as object receiver of method calls and member access expressions. However, the programmer is allowed to declare any variable or method parameter with the name it so that one can define a custom implicit object receiver in any scope of the program. Furthermore, when a lambda is expected to have a single parameter, the parameter can be omitted and it will be automatically available with the name it.

The Xbase syntax is extended in Edelta with a special syntax for accessing Ecore elements in a statically typed way, using ecoreref(…). In fact, Edelta programs refer directly to Ecore model classes, thus you do not need to access the Java code generated from an Ecore model. References to Ecore elements, such as packages, classes, data types, features and enumerations, can be specified by their fully qualified name in Edelta using the standard dot notation, or by their simple name if there are no ambiguities (possible ambiguities are checked by the compiler).

The Edelta editor (Sect. 3.2) provides typical Eclipse tooling mechanisms, including content assist and navigation to definitions. This is implemented both for Java types and for Ecore model elements. Navigating to Ecore model elements automatically opens the standard EMF Ecore tree editor.

All these features of the Edelta DSL support the requirements R1 and R3 discussed in the previous section.

3.2 The Edelta compiler, interpreter and IDE

In this section, we describe the main features of the Edelta compiler and its interaction with the Edelta interpreter, which provides a live and up-to-date view of the metamodels being modified in each specific part of the program. These characteristics implement what we have indicated in R3 and R5. Moreover, we briefly present the main features of the Edelta Eclipse integration.

One of the main distinguishing features of the Edelta compiler is that it automatically keeps track of the metamodels modified by the program according to the operations that are specified in the modifyEcore operations of the program. This makes Edelta an operation-based approach. The Edelta compiler interprets on-the-fly such operations during parsing and validation, including possible executions of Edelta reusable operations and calls to external Java code (via reflection, i.e., also in binary format). All the evolution operations are applied to an in-memory copy of the Ecore models imported into the program during the interpretation. This way, while the developer is writing an Edelta program in the Eclipse Edelta editor, the new and modified elements are immediately available: they can be referred using ecoreref, and they are used for static type checking. This is what we call *Live*

[2] For example, terminating semicolons are optional, the parenthesis can be omitted in a method call expression if the method has no parameter, and the return keyword is also optional in the body of an Edelta operation: the last expression will be returned.

Evolution. This particularly supports R2 since modelers do not need to wait to execute an Edelta program to see the effect, guiding and supporting a qualitative refactoring process. The semantics of the interpretation is the same as the semantics of the Java code generated by the Edelta compiler. Thus, if the Edelta compiler raises no validation errors by relying on the interpretation, then the Ecore models, after the modifications performed by the generated code, will still be valid Ecore models. This assures what we have identified in R3, i.e., metamodels manipulated with Edelta do not need a model repair process after the execution. For example, let us consider the following snippet of Edelta code:

```
1 modifyEcore ... {
2     addNewEClass("ANewClass")
3     // correctly resolved and type-checked
4     ecoreref(ANewClass).abstract = true
5 }
```

Since the compiler interprets the operations during validation and type checking, it can correctly resolve the ecoreref references, and it is also able to correctly verify that the last expression is correct, since it knows that ecoreref(ANewClass) is an EClass.[3]

The on-the-fly interpretation is applied sequentially to all expressions of the modifyEcore operations. This way, the Edelta compiler can keep track of the current state of the Ecore models being modified in each specific context of the program. The compiler can then detect possible errors due to references to stale elements in a particular part of the program. The compiler can issue meaningful error messages (instead of generic "unresolved reference" errors). The Edelta IDE provides helpful quick fixes for such error situations (e.g., it can propose using the new name of a previously renamed element).

In the context of an IDE, for most of the time, while the programmer is writing code in the editor, the program is invalid. An editor in an IDE continuously validates the program, and it must deal with error recovery: a validation error in a part of the program must not lead to many cascading errors in the rest of the program. In our context, we also have to keep error recovery into consideration when interpreting the program on the fly. If the interpreter evaluates an expression that contains an error, an exception will be thrown during such an evaluation. The Edelta interpreter takes care of handling such exceptions gracefully and tries to interpret as much as possible. Thus, expressions that change the Ecore under evolution will also be executed in the presence of program errors, and references to new or modified elements in the rest of the program will be resolved.

The Edelta editor provides typical Eclipse IDE features, like content assist, navigation to definitions, and quick fixes. These mechanisms are available both for Java types and for Ecore model elements. Because of the continuous interpretation, the content assist considers elements added or renamed in the program in a specific context. This way, it will propose only Ecore elements valid in a particular context of the program. Thus, the programmer does not risk introducing errors (e.g., access to renamed or removed elements) by respecting the operational safety in R3. R3 is also supported in case of parallel evolution of inter-dependent metamodels. This feature of Edelta is described in [54], where a consistency checking mechanism and an automatic code generation fully supports operational safety.

Keeping the in-memory Ecore model continuously updated by interpreting the specified evolving operations allows us to provide the developer with a quick view of the modified Ecore model in the Outline view. New elements created in the program or modified appear in the Outline view. Clicking on such an element in the Outline view tree highlights the expression responsible for the creation or modification of that element in the editor. As we will show in the next section, the Outline view of the Eclipse IDE shows the preview of the evolved metamodels, with touched elements in boldface.

Finally, thanks to Xbase, Edelta programs can be directly debugged while running the generated Java code in the Eclipse debugger: all the standard Eclipse debugging features, e.g., the views "Variables", "Breakpoints" and "Expressions" are available directly on the Edelta source code. While debugging the generated Java code is still possible, we believe that debugging the original Edelta source code is crucial for enhancing productivity.

3.3 Runtime library

The Edelta runtime library provides a set of Java APIs that implement the main basic refactoring operations that are applied on an Ecore model loaded in memory and also take care of saving the changed Ecore models into new files (Sec. 3.3.1). Besides the advantages of the presented Edelta infrastructure, another distinguishing feature is its *dynamic extensibility* concerning constraint validation as presented in Sec. 3.3.2.

3.3.1 Basic evolution operations

Edelta's APIs leverage the standard EMF runtime APIs but aim at improving them under many respects.

First of all, our APIs for creating Ecore elements are based on required features for such elements, so that one does not risk to create Ecore elements that would make the Ecore model invalid, i.e., respecting R3. For example, an EAttribute requires its feature eType to be set to an EDataType (though the type of the corresponding Java field, inherited

[3] This interpretation mechanism also works when using standard Java EMF API, e.g., by creating the EClass using the EcoreFactory.

from ETypedElement, is of supertype EClassifier). By using the standard EMF API, it is easy to create an invalid model by forgetting to call setEType or by passing an EClassifier that is not an EDataType:

```
EClass myClass = ...;
EAttribute a = EcoreFactory.eINSTANCE.createEAttribute();
a.setName("myAttr");
a.setEType(myClass); // valid Java code but invalid Ecore model!
```

Note that in the code above we also have to remember to add the created attribute to the container (the EClass). Forgetting to do that when creating Ecore complex models with references can easily lead to a model that cannot be saved due to some dangling references.

A few methods of the Edelta runtime library are shown, in a simplified form, in Listing 1.

```
public EAttribute addNewEAttribute(EClass eClass, String name,
    EDataType dataType, Consumer<EAttribute> initializer) {
  EAttribute attr = EcoreFactory.eINSTANCE.createEAttribute();
  attr.setName(name);
  attr.setEType(dataType);
  addEAttribute(eClass, e);
  initializer.accept(attr);
  return attr;
}

public EAttribute addNewEAttributeAsSibling(EStructuralFeature sibling,
    String name, EDataType dataType, Consumer<EAttribute> initializer) {
  return addNewEAttribute(sibling.getEContainingClass(),
      name, dataType, initializer);
}

public void addEAttribute(EClass eClass, EAttribute eAttribute) {
  addEStructuralFeature(eClass, eAttribute);
}

public void addEStructuralFeature(EClass cl, EStructuralFeature feature) {
  List<EStructuralFeature> features = cl.getEStructuralFeatures();
  EStructuralFeature existing = cl.getEStructuralFeature(feature.getName());
  checkAlreadyExistingWithTheSameName(cl, feature, existing);
  features.add(feature);
}
```

Listing 1 Some methods in the Edelta runtime library.

Thus, we provide a method for creating a new EAttribute and adding it into an EClass (which can be specified directly or taken from an existing sibling feature), adding a given EAttribute into an EClass, and, in general, adding a given EStructuralFeature into an EClass. These methods rely on each other, and, most of all, they both rely on checkAlreadyExistingWithTheSameName, shown in Listing 2. That method makes sure that the addition does not make the model invalid. In these simple addition operations, such a validation step just checks possible existing elements with the same name in the given container. This relies on the constraint validation mechanism described in Sect. 3.3.2. In particular, we throw an IllegalArgumentException so that the interpreter stops the execution of the single expression that led to the exception: as described in Sect. 3.2, the interpreter will keep on interpreting and validating the rest of the program, thanks to its error recovery mechanisms.

```
private <C extends ENamedElement, E extends ENamedElement> void
    checkAlreadyExistingWithTheSameName(C container, E toAdd, E existing) {
  if (existing != null) {
    String errorMessage = getEObjectRepr(container) +
        " already contains " +
        existing.eClass().getName() + " " +
        getEObjectRepr(existing);
    showError(toAdd, errorMessage);
    throw new IllegalArgumentException(errorMessage);
  }
}
```

Listing 2 The method for ruling out duplicates.

All the methods that add an element to a container have a similar implementation, ensuring that a valid metamodel stays valid after the evolution. The Edelta runtime consists of several methods of the above shape, including overloaded versions (for example, when the initializer is not needed). Similarly, we have methods for adding a superclass (addESuperType) and for removing elements from a container (removeEClass, removeEAttribute, etc.). These checks will be seamlessly reused when we write more complex refactorings.

Most methods for adding and creating elements in our Edelta runtime Java library have this shape, returning the element that has been created and added to a container. This allows the client code to use method chaining for easily creating complex models. Moreover, a lambda can be passed as the last argument to further initialize the created element (the parameter initializer of type Consumer). Note that such a lambda is executed only after the element has been added to the container. This also makes sure that the model has no dangling references (see, instead, the Java code above using directly the EMF API).

We also provide the operation removeElement for removing types and features. Again, such operations make sure that after the modification the Ecore model is still valid. This requires removing also possible references to the removed element. This is particularly close to what integrity constraint checks apply in the context of database [55]. For example, the result of the removal listed in Listing 3 applied to the metamodel in Fig. 1 is shown in Fig. 2. Note that besides removing

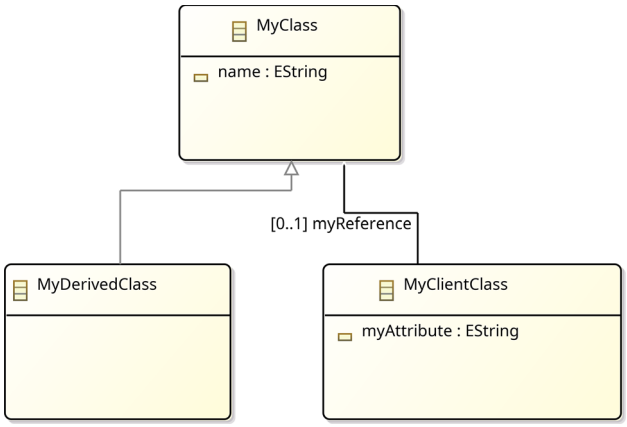


Fig. 1 Base metamodel

the specified element MyClass, this refactoring also removes the relations involving the removed class, like the inheritance relation of MyDerivedClass, and the reference myReference of type MyClass.

```
1 removeElement(ecoreref(myecore.MyClass))
2 // or using extension methods
3 ecoreref(myecore.MyClass).removeElement
```

Listing 3 Edelta example of removing a class.

On top of the basic operations like the ones shown above, we also provide composed operations for recurring tasks, like adding a reference (or attribute) that is required or making a reference a containment bidirectional reference. These definitions are straightforward in the Edelta DSL, as shown in Listing 4 where we omit other utility operations. This way, defining a complex refactorings like the ones shown in the next sections is much easier and the complex refactoring operations will also be much more readable.

```
1 def addMandatoryReference(EClass eClass, String
      ↪ referenceName, EClass type) {
2     return eClass.addNewEReference(referenceName, type) [
3         makeSingleRequired
4     ]
5 }
6 def makeContainmentBidirectional(EReference reference) {
7     reference.makeContainment
8     val owner = reference.EContainingClass
9     val referredType = reference.EReferenceType
10    referredType.addMandatoryReference(owner.
          ↪ fromTypeToFeatureName, owner) => [
11        makeBidirectional(reference)
12    ]
13 }
```

Listing 4 Edelta utility functions

Although these operations are compositions of basic operations, we still consider them as basic refactoring operations and we include them in the standard runtime library of Edelta. We believe that providing these functions with meaningful names, even if some cases they consist of a single instruction,

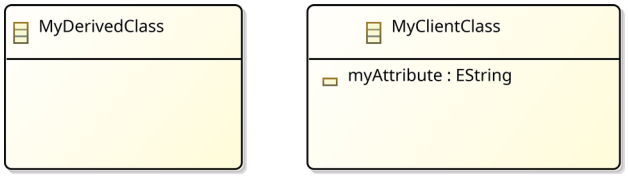


Fig. 2 After removal

will make writing more complex evolution functions much easier and will make such functions very readable and easily maintainable. We also think that this approach makes implementing refactoring operators much less error prone than to deal with standard EMF API directly.

The Edelta Java API can be used directly in Java for creating and modifying Ecore models, using method chaining and lambdas for initializing elements, relying on the method signatures that force the client to specify all required features. For example, we can create an EClass with an EAttribute of type EString, with an explicit lower-bound, like this, assuming ePackage is an EPackage:

```
1 addNewEClass(ePackage, "MyClass",
2   c -> addNewEAttribute(c, "myAttr", EcorePackage.
        ↪ eINSTANCE.getEString(),
3     a -> a.setLowerBound(2)
4   )
5 );
```

However, using the Edelta DSL, the same operations can be expressed more easily and in a more readable way as follows:

```
1 ePackage.addNewEClass("MyClass") [
2   addNewEAttribute("myAttr", ecoreref(EString)) [
3     lowerBound = 2
4   ]
5 ]
```

In the above code, we show many linguistic mechanisms of the Edelta DSL, namely, extension methods, less verbose lambda syntax, implicit lambda parameter (it) and syntactic sugar for setter methods, not to mention the special syntax ecoreref to refer an Ecore element.

The runtime library that we have just described is automatically available in any Edelta programs. Thus, no further specific configuration is required: the API that we have illustrated can be seamlessly used both in reusable operations and in modifyEcore operations.

3.3.2 Constraint validation

Edelta users can participate in the validation phase of the compiler and interpreter without having to customize the Edelta implementation itself. The standard Edelta runtime library provides two methods, showError and showWarning, which the Edelta users can call in their own Edelta code to "show" errors and warnings, respectively, also specifying the cause of the issue (typically an Ecore model element).

Table 3 The complete refactoring catalog

ID	Refactoring	Op
R1	Add package	R2
R2	Remove Package	R1
R3	Create Metaclass	R4
R4	Remove Metaclass	R3
R5	Add Attribute	R7
R6	Add Reference	R8
R7	Remove Attribute	R5
R8	Remove Reference	R6
R9	Set Reference opposite	R10
R10	Unset Reference opposite	R9
R11	Add Datatype	R12
R12	Remove DataType	R11
R13	Create Enum	R14
R14	Remove Enum	R13
R15	Add Literal	R16
R16	Remove Literal	R15
R17	Merge Literals	R18
R18	Split literal	R17
R19	Rename Package	
R20	Rename Metaclass	
R21	Rename feature	
R22	Change Package	
R23	Make MC Abstract	R24
R24	Make MC concrete	R23
R25	Add superclass	R26
R26	Remove superclass	R25
R27	Make ref composition	R28
R28	Make ref aggregation	R27
R29	Generalize attribute	R30
R30	Specialize attribute	R29
R31	Generalize Reference	R32
R32	Specialize Reference	R31
R33	Generalize Supertype	R34
R34	Specialize Supertype	R33
R35	Pull Up Attribute	R36
R36	Push Down Attribute	R35
R37	Pull Up Reference	R38
R38	Push Down Reference	R37
R39	Extract Superclass	R40
R40	Inline superclass	R39
R41	Fold superclass	R42
R42	Unfold superclass	R41
R43	Extract Subclass	R44
R44	Inline Subclass	R43
R45	Extract Metaclass	R46
R46	Inline Metaclass	R45
R47	Fold Metaclass	R48
R48	Unfold Metaclass	R47
R49	Move feature over ref	R50
R50	Collect features over ref	R49
R51	Subclasses to Enum	R52
R52	Enum to Subclasses	R51
R53	Merge attrs	R54
R54	Split attr	R53
R55	Merge refs	R56
R56	Split ref	R55
R57	Merge metaclasses	R58
R58	Split metaclass	R57
R59	merge enumerations	R60
R60	split enum	R59

The standard Edelta interpreter catches such method calls and uses the arguments of such method calls to create errors and warnings, respectively, on the program.

Thus, this mechanism allows the programmer to extend the validation aspect of Edelta on the fly, turning the Edelta IDE into a live development environment. Indeed, the user can easily experiment with such validation mechanisms even without saving the file. Besides letting the users extend the Edelta validations, the mechanism described above also allows us to easily extend Edelta without hardcoding checks in the compiler.

For example, in [22] we presented our Edelta reusable library of metamodel bad smell finders and the corresponding refactorings. The bad smell finders have been modified so that they use `showWarning` when they spot a bad smell.[4] Thus, the users can call our bad smell finders from within their Edelta programs and can see live whether their metamodels under modifications have some bad smells. Moreover, they can make sure that, while they evolve their models in the current program, they do not introduce other bad smells. This particularly supports R2.

4 A metamodel refactoring catalog

Starting from the basic refactorings, we built more complex refactorings. In the following, we report some of the most used refactorings in the literature, with exemplary Edelta code and the initial and resulting metamodels. We refer to the Edelta source code[5] for the complete catalog and summarized in Table 3.

[4] We prefer to only show a warning instead of an error in case of a bad smell. Indeed, the Edelta compiler does not generate Java code if a program has an error. On the contrary, it still generates code in case of warnings.

[5] https://github.com/LorenzoBettini/edelta.

Fig. 3 Base metamodel

4.1 Extract class

This refactoring is applied when a class becomes overweight with too many features and its purpose becomes unclear. *Extract Class* involves creating a new class and moving a given collection of features from the old one to the new one. Being this a complex refactoring, it uses other basic operations, e.g., move attribute/reference, add class. In the containing class, a containment required bidirectional reference to the extracted class will be created (its name will be the name of the extracted class with the first letter lowercase), as seen in [28]. This containment reference will make sure that the new introduced class will be contained in the Ecore resource. The definition in Edelta is shown in Listing 5. Note how we use some utility functions as the ones shown in Listing 4, Sect. 3.3.1. Again, we use extension methods a lot in the definition.

```
def extractClass(String name, Collection<
    ↪ EStructuralFeature> features) {
  checkNoBidirectionalReferences(features,
      "Cannot extract bidirectional references")
  val owner = findSingleOwner(features)
  val extracted = owner.addNewEClassAsSibling(name)
  val reference = owner.addMandatoryReference(name.
      ↪ toFirstLower, extracted)=>[
    makeContainmentBidirectional
  ]
  features.moveAllTo(extracted)
  return reference
}
```

Listing 5 Edelta definition of Extract Class

The functions we implement in Edelta typically return information about the change that took place. In the case of extractClass, we return the (containment bidirectional) reference to the class that has been created (the variable extracted in the listing). This returned reference can be used both to access the containing class that has been modified (since its features have been extracted) and the newly created class.

This example of complex refactoring also highlights another important aspect: some refactorings can be safely applied only if some specific conditions hold. For example, in the above extractClass we must make sure that the features that need to be extracted do not contain any bidirectional reference. In the presence of a bidirectional reference, this refactoring would result in an invalid metamodel. In the presence of a bidirectional reference, the called checkNoBidirectionalReferences (not shown here) will raise an error and will terminate the function without touching the metamodel.

Moreover, we avoid requiring redundant information that can be easily computed from within the definition. For example, in extractClass we only require the name for the new class and the collection of features to extract. We assume that the features all belong to the same class, and we retrieve such a class with our utility function findSingleOwner. Once again, if we cannot find such a unique owner we terminate the function with a meaningful error. Both checks rely on the mechanism described in Sect. 3.3.2.

In Listing 6, we show an example of invocation of the above refactoring. We use the metamodel in Fig. 3, and the resulting metamodel is shown in Fig. 4. The invocation of extractClass will return the EReference workAddress.

```
extractClass("WorkAddress", #[
    ecoreref(Person.street),
    ecoreref(Person.workplace),
    ecoreref(Person.houseNumber)
])
```

Listing 6 Edelta example of use of Extract Class

4.2 Extract superclass

When two or more classes are meant for similar tasks, we have a kind of concept duplication. A refactoring for simplifying such situations via inheritance introduction is *Extract Superclass*. Indeed, often this duplication remains unnoticed until classes are created, thus requiring an inheritance relation to be introduced later [56,57]. The definition of this refactoring in Edelta is shown in Listing 7.

```
def extractSuperclass(String name, List<EStructuralFeature
    ↪ > duplicates) {
  val feature = duplicates.head;

  return feature.EContainingClass.addNewEClassAsSibling(
      ↪ name) [
    makeAbstract
    duplicates
      .map[EContainingClass]
      .forEach[c | c.addESuperType(it)]
    pullUpFeatures(duplicates)
  ]
}
```

Listing 7 Edelta example of Extract Superclass

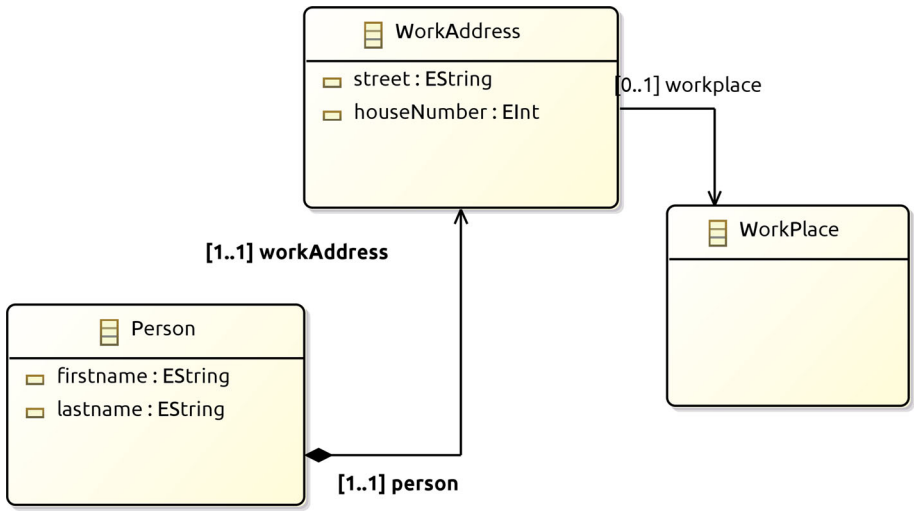


Fig. 4 After extract class

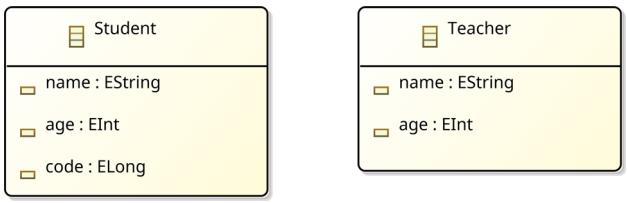


Fig. 5 Base metamodel

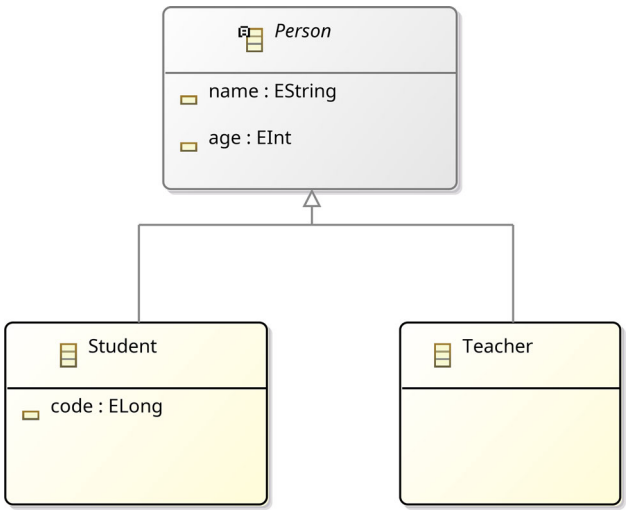


Fig. 6 After extract superclass

Note that the introduced superclass is made abstract. Apparently, no check is performed on the features to extract: are they all the same feature? That is, do they have the same name, type and other equal characteristics (e.g., required, multiplicity, etc.)? Actually, this is all delegated to another refactoring described in the next section. This also shows that some complex refactorings can be, in turn, defined in terms of other complex refactorings.

4.3 Pull up

With *Pull Up*, we remove a list of features that are duplicate in their classes and we "pull them up" into a superclass. The definition of such a refactoring in Edelta is shown in Listing8. This operation is void since there's no need to return anything to the caller.

```
1 def pullUpFeatures(EClass dest, List<EStructuralFeature>
      ↪ duplicates) : void {
2     checkNoDifferences(
3         duplicates,
4         new EdeltaFeatureDifferenceFinder().
              ↪ ignoringContainingClass,
5         "The two features are not equal"
6     )
7     checkAllDirectSubclasses(dest, duplicates.map[
          ↪ EContainingClass])
8     duplicates.head.copyTo(dest)
9     removeAllElements(duplicates)
10 }
```

Listing 8 Edelta example of Pull Up

Note that, as usual, we perform a few checks. In particular, as anticipated in the previous refactoring, we must make sure that all the passed features are equal, of course, not considering their containing class that will be different (this is implemented by using our EdeltaFeatureDifferenceFinder and its fluent API). Moreover, we also make sure that the containing classes of such features are all direct subclasses of the destination class. Finally, it is easy to verify that the definition of extractSuperclass in Listing 9 respects the constraints of pullUpFeatures of Listing 8.

In Listing 9, we show an example of invocation of the above refactorings. We use the metamodel in Fig. 5, and the resulting metamodel is shown in Fig. 6. We first extract a superclass, Person, with the features names, and then, we also pull-up the other two common features ages. Note that thanks to our Edelta on-the-fly interpretation during the

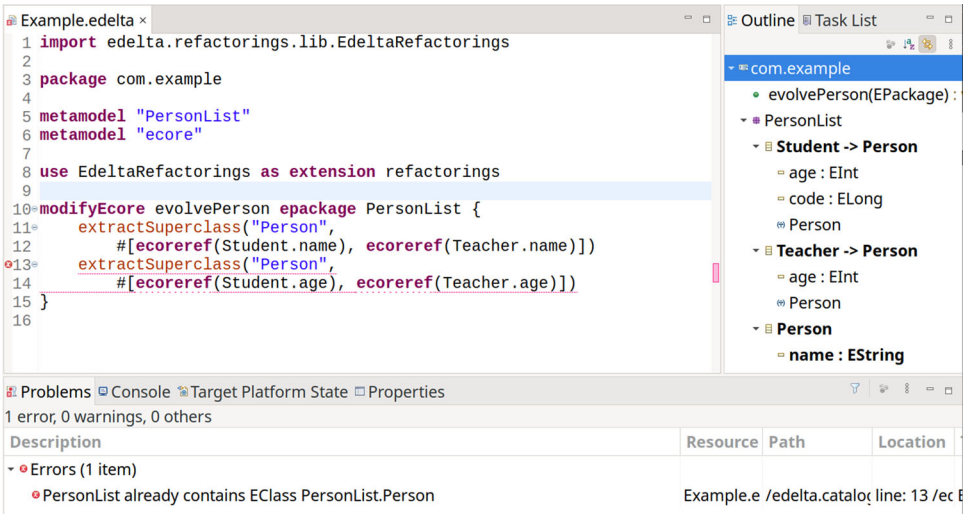


Fig. 7 An invalid double application of extractSuperclass

compilation (Sect. 3.2), after the execution of extractSuperclass, the newly introduced superclass Person can be directly referred with ecoreref in the rest of the program.

```
1 extractSuperclass("Person",
2     #[ecoreref(Student.name), ecoreref(Teacher.name)])
3 pullUpFeatures(ecoreref(Person),
4     #[ecoreref(Student.age), ecoreref(Teacher.age)])
```

Listing 9 Edelta example of use of Extract Superclass

If we performed the same refactoring extractSuperclass twice, we would get an error from the Edelta compiler, since the second application would try to add a new class in the package with the same name of an existing one, "Person." This can be seen in Fig. 7 where we simulated this situation. The error is raised when extractSuperclass (Listing 7) calls addNewEClass of our standard Edelta library of basic refactorings (Sect. 3.3.1). The method addNewEClass (which has a shape similar to addNewEAttribute shown in Listing 1) uses the method shown in Listing 2, making sure that classes in the same package have different names. That method, in turns, relies on the constraint validation mechanism described in Sect. 3.3.2 and the Edelta IDE (Sect. 3.2) automatically notifies the developer with error markers. Thus, when writing complex refactorings as extractSuperclass we can simply rely on the existing basic refactorings to take care of not making an existing valid metamodel invalid, and we only have to focus on writing additional constraint checks that are specific of the new refactoring we are implementing.

Figure 7 shows another interesting feature of the Edelta compiler: its interpreter can still apply valid operations to the in-memory model. This is based on the error recovery mechanism described in Sect. 3.2. The Outline View shows that the first valid call to extractSuperclass has been successfully applied, in spite of the whole program not being valid. In boldface, we can see the new introduced base class Person with the extracted feature name and the introduced "extends" relation in Student and Teacher (also in this case, the boldface highlights a modification in the model). In fact, our compiler is able to recover from possible errors raised during the interpretation, so that the modeller can still have a view of the current state of the model after valid operations have been applied.

4.4 Opposite refactorings

In Table 3, we also reported the opposite refactorings when applicable. In this section, we will show an example of opposite refactoring. The refactoring *Inline Class* is the opposite of *Extract Class* whose implementation in Edelta, extractClass, was showed in Listing 5. We show the implementation of *Inline Class* in Edelta, inlineClass in Listing 10.

```
1 def inlineClass(EClass cl) {
2     val ref = findSingleContainmentReferenceToThisClass(cl
          ↪ )
3     checkNotMany(ref,
4         "Cannot inline in a 'many' reference")
5     val featuresToInline = cl.EStructuralFeatures
6         .filter[it != ref.EOpposite] // skip the possible
              ↪ back reference
7     featuresToInline.moveAllTo(ref.EContainingClass)
8     removeElement(cl)
9     return featuresToInline
10 }
```

Listing 10 Edelta definition of Inline Class

Indeed, this definition basically performs the opposite operations of the ones shown in Listing 5. In particular, it searches for one (and only one) containment reference to the class that has to be inlined (of course, it terminates with an error if such a search fails). In fact, the containment reference will be used to inline the features of the passed class into the owner class of such a containment reference, ref.EContainingClass. (The extractClass of in Listing 5 intro-

Fig. 8 Extract class

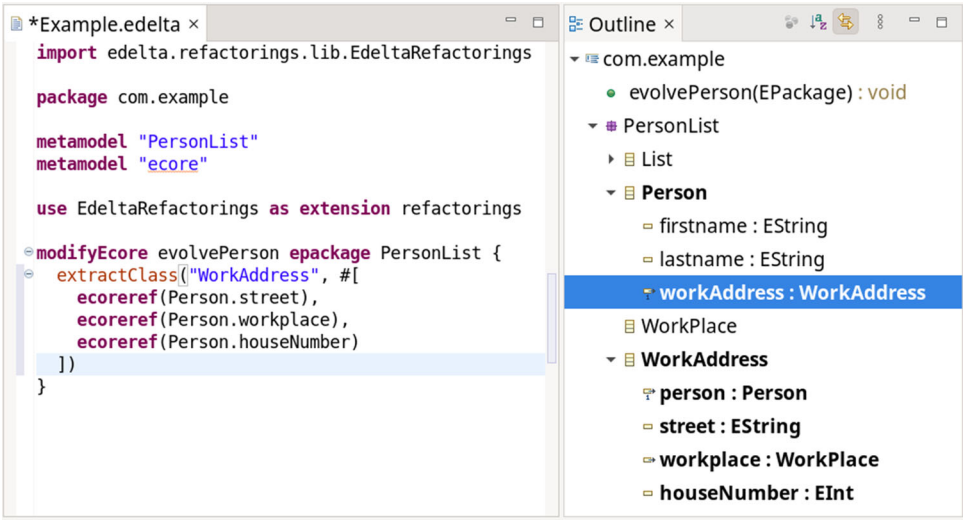
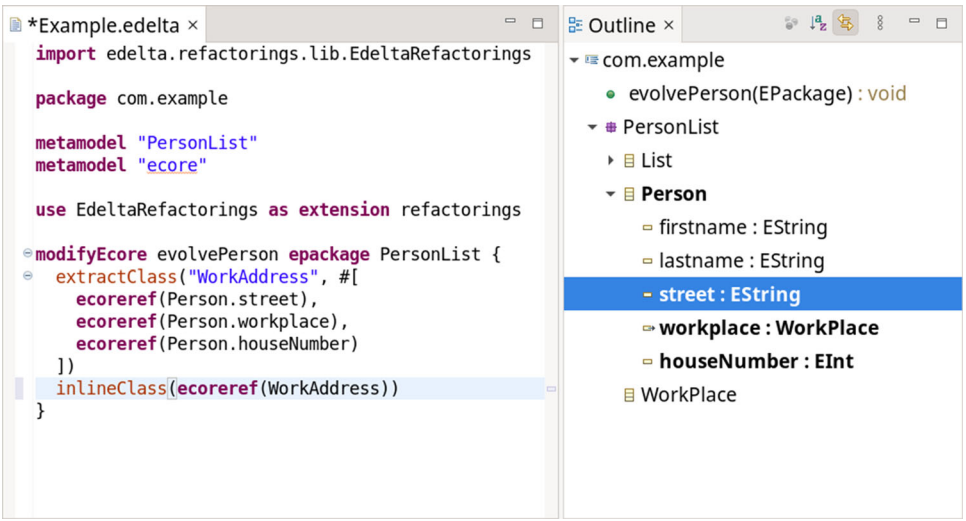


Fig. 9 Inline class after extract class



duced such a containment reference.) The move will concern all the features except for the possible opposite reference of such a containment reference (our extractClass introduces a bidirectional containment reference, but the fact that it is bidirectional is not strictly required). This containment reference (with its possible opposite reference) will also be automatically removed when calling our removeElement simple operation, as shown in Sect. 3.3.1.

It is straightforward to verify that if we successfully apply extractClass to a metamodel, we can also successfully apply inlineClass to the resulting metamodel, obtaining exactly the same original metamodel. That is, if we execute the snippet of Listing 11 by using the metamodel in Fig. 4, we obtain the metamodel of Fig. 3. Of course, also the other way round holds.

```
1 inlineClass(ecoreref(WorkAddress))
```

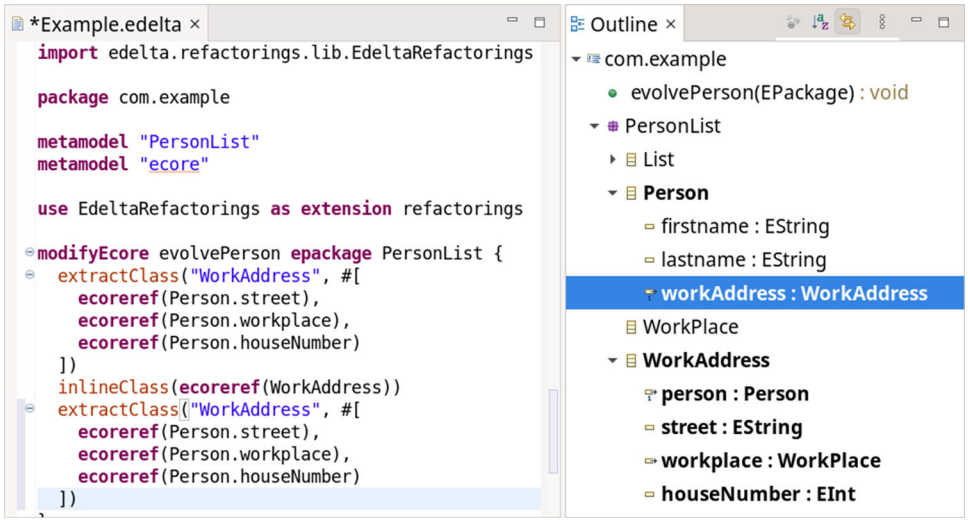
Listing 11 Edelta example of use of Inline Class

In particular, we have a extensive test suite for our refactorings, and we also verify the above properties for the refactorings that have an opposite (see Sect. 5). Basically we make sure that we can always invert a refactoring through its opposite, if it has one.

For most refactoring functions in our catalog, we provide some variants in overloading, in order to specify additional arguments to replace some default values. For example, we also provide an overloaded inlineClass taking also the prefix for the names of the features to inline.

We also use this example of opposite refactorings to further show the Edelta Eclipse editor live capabilities (Figs. 8, 9, 10), already mentioned in Sect. 3.2. Note that the file is not even saved, but the interpreter used by the compiler allows us to always show in the Outline an up-to-date state of the metamodel being evolved. The outline also shows in boldface the elements that have been changed (added or modified, e.g., moved). The metamodel being refactored is the one of Fig. 3. In Fig. 8, we apply extractClass (and the outline shows

Fig. 10 Extract class after inline class



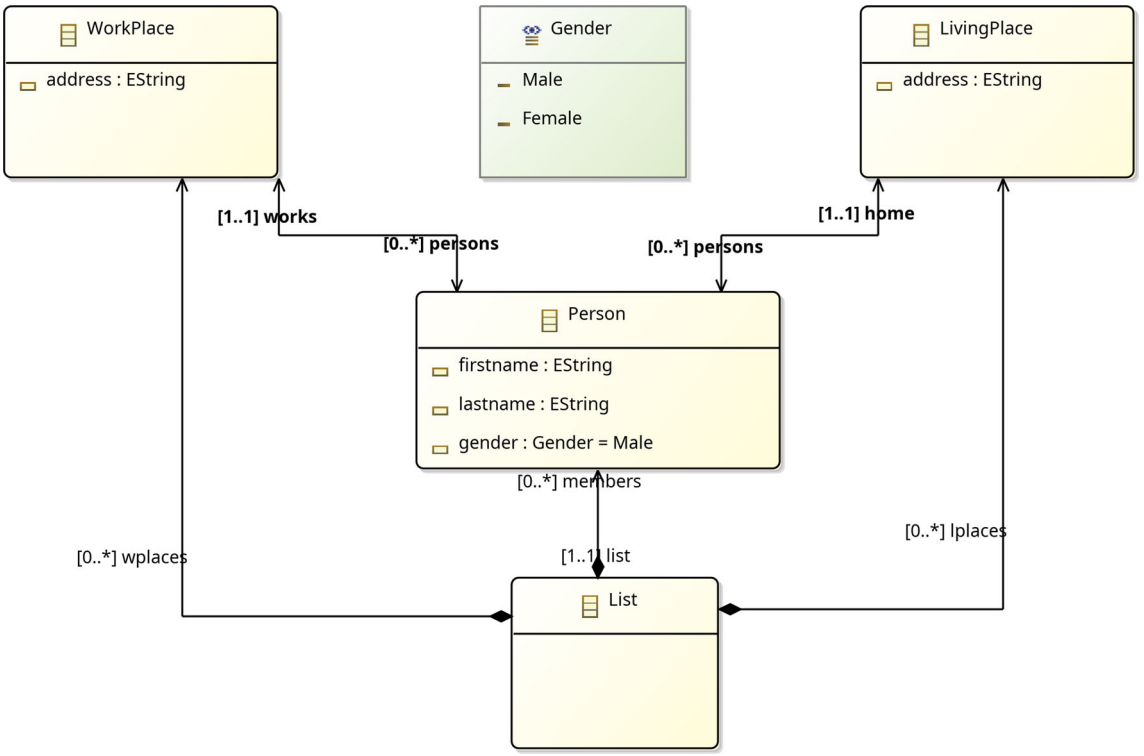


Fig. 11 The original PersonList metamodel

the new version of the metamodel); then (Fig. 9), we apply the opposite refactoring inlineClass, obtaining the original metamodel of Fig. 3; finally (Fig. 10), we apply extractClass again, going back to the metamodel shown in the Outline of Fig. 8. Of course, the example is intentionally fictitious.

4.5 Using the catalog

The implemented operators of the proposed catalog are reported in Table 3. This table has been derived by the refactorings presented in [28], and all the refactorings have been implemented with Edelta and made available to the community.[6]

Since the basic refactoring operations are part of the Edelta runtime library, are automatically available in any Edelta program. On the contrary, the catalog of complex refactorings illustrated in this section is included in a separate project, called edelta.refactorings.lib. Thus, to use the catalog in an Edelta Eclipse project, the Java bundle edelta.refactorings.lib has to be added as a dependency in

[6] Just like in [28] we are not considering elements that cannot be instantiated in models: annotations, derived features and operations.

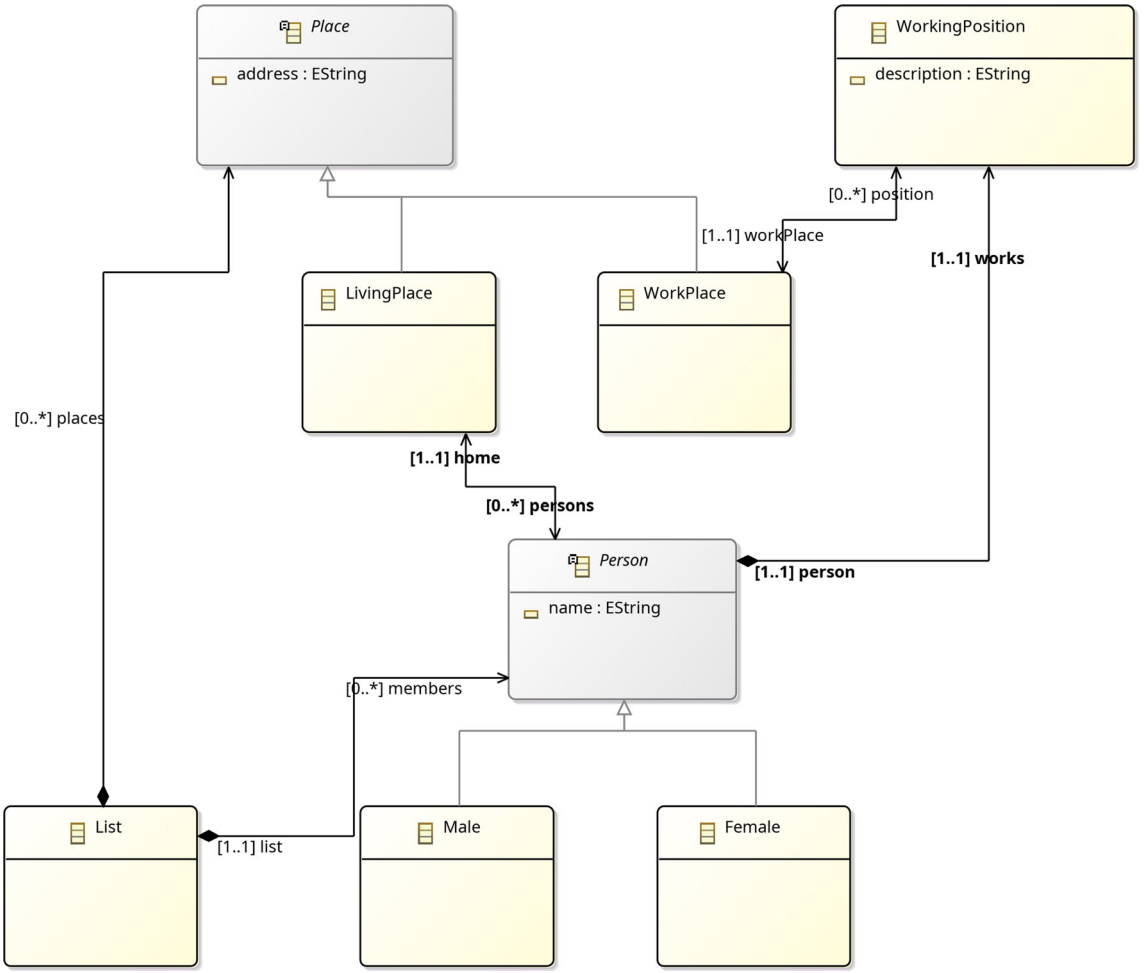


Fig. 12 The evolved PersonList metamodel

the MANIFEST.MF.[7] Then, in the Edelta file, the EdeltaRefactorings has to be imported (Java-like import) and "used" in the program with the clause use …as … (Sect. 3.1).

In Listing 12, we show an example that uses the catalog. This program applies a few refactoring operations to the metamodel shown in Fig. 11. The result of such operations is shown in Fig. 12.

```
1 import edelta.refactorings.lib.EdeltaRefactorings
2
3 package edelta.personlist.example
4
5 metamodel "PersonList"
6 metamodel "ecore"
7
8 use EdeltaRefactorings as refactorings
9
10 modifyEcore improvePerson epackage PersonList {
11     refactorings.enumToSubclasses(ecoreref(Person.gender))
12     refactorings.mergeFeatures("name",
13         #[ecoreref(Person.firstname), ecoreref(Person.
              ↪ lastname)])
14 }
15
16 modifyEcore introducePlace epackage PersonList {
17     refactorings.extractSuperclass("Place",
18         #[ecoreref(LivingPlace.address), ecoreref(
              ↪ WorkPlace.address)])
19 }
20
21 modifyEcore introduceWorkingPosition epackage PersonList {
22     refactorings.referenceToClass("WorkingPosition",
          ↪ ecoreref(Person.works)) => [
23         addNewEAttribute("description", ecoreref(EString))
24     ]
25     ecoreref(WorkPlace.persons).name = "position"
26 }
27
28 modifyEcore improveList epackage PersonList {
29     refactorings.mergeFeatures("places",
30         ecoreref(Place),
31         #[ecoreref(List.wplaces), ecoreref(List.lplaces)]
32     )
33 }
```

Listing 12 Example using the catalog

[7] As said in Sect. 3.1, we also deploy Edelta projects as Maven artifacts to the Maven Central repository; when using Edelta in a Maven project, the edelta.refactorings.lib has to be added as a Maven dependency.

Besides extractSuperclass (Sect. 4.2), in this example we use several other refactorings of the catalog, that are not explained in details, but their meanings and usage should be straightforward and should be easily understood by looking at the original metamodel (Fig. 11) and the evolved one (Fig. 12).

5 Contributing to the catalog

In this section, we show how requirements R4, and consequently R6 are respected on the proposed project.

Edelta development highly relies on automated tests. We try to use plain and fast JUnit tests as much as possible. For example, we can test our runtime library and our catalog with plain JUnit tests and this allows us to run the full test suite of the runtime library and catalog, which consists of about 250 tests, in about 2 s.[8]

Concerning the DSL and its compiler (which also relies on the interpreter), core aspects related to the language itself (e.g., parsing, validation and type system and code generation) are also tested with unit tests that do not use the Eclipse user interface. Again, this allows to have a very fast test suite: about 400 tests run in less than 20 s. Of course, aspects related to the Eclipse IDE tooling (e.g., content assist, navigation, automatic building) are also tested with integration tests, where the core aspects are integrated with the Eclipse environment. The integration tests, as expected, take much longer to execute, since they start a full Eclipse environment: about 60 integration tests take about 1 min.

Moreover, we take into consideration many code quality metrics, like code coverage (i.e., how many parts of the Edelta code base are executed by tests), possible bugs, code smells and other metrics like code duplication. All these test oriented development mechanisms are part of a continuous integration process, relying on free cloud services. The code of Edelta is hosted on GitHub and for each commit, an automatic build process is started on GitHub Actions. This automated build process compiles all the Edelta modules and runs all the tests (unit and integration tests, including end-to-end tests against the Eclipse distribution we make available to our users). In particular, Edelta is built and tested in all the virtual environments provided by GitHub Actions, that is, at the time of writing, Linux, Windows and macOS. We also make sure we can build it and test with Java 11 (the minimal required version) and with Java 17 (the current LTS version). Moreover, our build on GitHub Actions also keeps track of code coverage (using the cloud service Coveralls) and analyzes the code quality metrics using SonarCloud (a cloud service based on the well-known code quality analysis tool SonarQube).

[8] These statistics are measured on typical modern computers with quad core, 16 Gb of RAM and SSD, running Linux.

Besides possible build or test failures, these cloud services also make the build fail in case code coverage goes beyond 100% and code quality metrics are not respected.[9] Although we are aware that such metrics are not a sufficient condition for the correctness of Edelta, we just want to stress how much we care for the quality of our software.

We rely on GitHub as the code hosting platform and as the Continuous Integration platform also for its native mechanisms for contributions from other users. In particular, our continuous integration infrastructure allows us to carefully review possible contributions. Under that respect we aim at providing some contribution guidelines.

If users want to contribute a new refactoring to our Edelta refactoring library, they just have to follow the typical workflow of GitHub (and in general of every Git hosting platforms):

- they fork our Edelta GitHub repository on their own GitHub account;
- they add a new refactoring to our library;
- they add tests for the new refactoring;
- they create a pull request (PR)

Upon creation of the PR, our continuous integration process automatically starts, making sure that everything still compiles and all tests succeed. Moreover, this process also makes sure that code coverage does not decrease. If that happens, the PR gets a negative feedback and the users are aware that their tests do not fully cover the new proposed refactoring. They are then encouraged to fix that and to update the PR, so that the continuous integration process starts again. Of course, we also review the PR ourselves. This allows us to start a conversation with the contributors and propose some possible change requests. Finally, the PR can be merged.

We now describe the procedure we follow ourselves when introducing a new refactoring to the catalog. We suggest to follow the same procedure for contributing new refactoring operations.

We start by using the Eclipse editor, with an input metamodel and use the live environment to see the modifications on the fly (Outline view). The input metamodel is constructed appropriately to be used as an input for that particular refactoring. The refactoring operation is written in an Edelta program as a reusable operation and in the same program we also write at least one modifyEcore operation acting on the input metamodel and calling the refactoring operation under development. As described in Sect. 3.2, the Edelta Eclipse editor, by using the on-the-fly interpreter, allows us

[9] The results of code quality analysis of Edelta on SonarCloud are available at https://sonarcloud.io/dashboard?id=io.github.lorenzobettini.edelta%3Aedelta.parent. We note that we aim at respecting all the quality gates of SonarCloud.

to have an immediate view of the evolved metamodel by simply looking at the Outline view (see the screenshots in Sect. 4). This implies fast roundtrips to quickly have a prototype of a refactoring. There's no need to run the generated Java code and manually inspect the evolved metamodel, since we can see the evolved metamodel live in the Outline view (we do not even have to save the Edelta file, since the Outline is always updated continuously and the Edelta interpreter keeps it always up-to-date, according to the operations called in the modifyEcores).

When we have a working prototype of the refactoring operation, we copy that operation to the Edelta file of the catalog (EdeltaRefactorings.edelta in the project edelta.refactorings.lib, see Sect. 4.5). Then, we add a few unit tests to the test case of the catalog (stored in the tests project edelta.refactorings.lib.tests). The metamodel .ecore file that we used for the prototype is stored in the tests project, in a subdirectory of the input metamodels directory.

We provide a few testing APIs and utility functions to write unit tests in a compact way. The main idea is that each test specifies the Ecore file to load, it calls the refactoring function with the appropriate arguments and then verifies that the evolved metamodel is as expected. The expected Ecore file corresponding to the input Ecore file is stored in a subdirectory of the expected metamodels directory. The evolved metamodel files are generated each time in another temporary directory, and these metamodel files are then compared to the corresponding ones in the expectations directory. The first time such a test is executed it will fail because there is no expected output file. We then manually inspect the generated output files, and if we are sure they are as we want, we copy the generated out files in the corresponding expectations directory.

These are a selection of unit tests of our catalog's test suite. In particular, these are testing inlineClass (Sect. 4.4):

```
1 @Test
2 void test_inlineClassWithAttributes() throws IOException {
3     withInputModels("inlineClassWithAttributes", "
          ↪ PersonList.ecore");
4     loadModelFiles();
5     refactorings.inlineClass(getEClass("PersonList", "
          ↪ Address"));
6     refactorings.saveModifiedEcores(AbstractTest.MODIFIED)
          ↪ ;
7     assertModifiedFiles();
8 }
9
10 @Test
11 void test_inlineClassWithAttributesWronglyMulti() throws
       ↪ IOException {
12     withInputModels("inlineClassWithAttributesWronglyMulti
          ↪ ",
13         "PersonList.ecore");
14     loadModelFiles();
15     assertThrowsIAE(() ->
16         refactorings.inlineClass(getEClass("PersonList", "
              ↪ Address")));
17     refactorings.saveModifiedEcores(AbstractTest.MODIFIED)
          ↪ ;
18     assertModifiedFilesAreSameAsOriginal();
19     assertThat(appender.getResult().trim())
20         .isEqualTo(
21         "ERROR: PersonList.Person.addresses: " +
22         "Cannot inline in a 'many' reference: PersonList.
              ↪ Person.addresses");
23 }
24
25 @Test
26 void test_inlineClass_IsOppositeOf_extractClass() throws
       ↪ IOException {
27     withInputModels("extractClassWithAttributes", "
          ↪ PersonList.ecore");
28     assertOppositeRefactorings(
29         () -> refactorings.extractClass("Address",
30             asList(
31                 getEAttribute("PersonList", "Person", "
                      ↪ street"),
32                 getEAttribute("PersonList", "Person", "
                      ↪ houseNumber"))
33             ),
34         () -> refactorings.inlineClass(getEClass("
              ↪ PersonList", "Address"))
35     );
36     assertLogIsEmpty();
37 }
```

Thanks to our testing API for setting up and verification, it is quite straightforward to write such tests, which are also maintainable and readable. We use a few utility methods getEAttribute, getEClass, …, to retrieve the input Ecore elements to pass to the refactoring function. We also use AssertJ[10] for readable and fluent assertions. In this example, we show a test case verifying a failure for inlineClass.

It is crucial to fully cover the refactoring function body: we test the "happy case" and the corner cases. For error situations, we have tailored metamodels recreating a context for which the refactoring cannot be applied. Note that in such a case, we verify the expected error message and we make sure the refactoring does not modify the metamodel at all (assertModifiedFilesAreSameAsOriginal). When we want to verify that no error/warning message is generated by a refactoring we can use the utility function assertLogIsEmpty.

In case of opposite refactorings (Sect. 4.4), we test this property as well, as shown in the last test shown in the above listing. Again, we provide an API to simply specify the two refactoring invocations (in the shape of a Java lambda expression) that are expected to be one the opposite of the other. The verification will then be carried out automatically: the first refactoring is applied to the input metamodel, the second refactoring is applied to the evolved metamodel and the resulting metamodel must be exactly the same as the original input metamodel.

[10] https://github.com/assertj/assertj-core.

6 Conclusions and future work

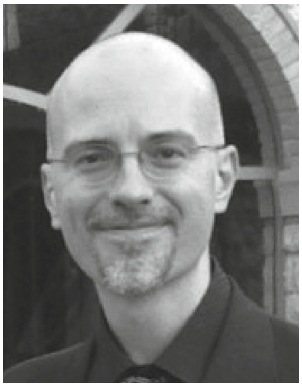
Metamodels are core assets constituting an ecosystem of model-based artifacts, that constantly change and evolve over time. Therefore, whenever a metamodel changes, all the interrelated artifacts must be accordingly adapted in order to remain valid. For this reason, supporting the metamodel evolution phase is fundamental not only for this stage, but also for enabling automation in the related activities, e.g., co-evolution. In this paper, we outlined the need for an executable metamodel refactorings catalog that should be open to contribution and maintained. We show how using the Edelta language, we built a complete catalog of nearly 60 refactorings, shared with the community, and we hope modelers can contribute by following the described process. As a future plan, we would like to migrate the Edelta compiler to the cloud to support the environment as a service. The advantage of this future version is that the modeler could use the refactoring compiler online and interact with the DSL in a standalone or web-based tool. We also started to extend Edelta in order to support also refactorings of metamodel instances. In this way, Edelta could support the co-evolution of models and metamodels using a single framework for evolution and model migration.

Funding Open access funding provided by Gran Sasso Science Institute - GSSI within the CRUI-CARE Agreement.

References

1. Opdyke, W.F.: Refactoring object-oriented frameworks, Ph.D. thesis, USA (1992)
2. Iwamoto, M., Zhao, J.: Refactoring aspect-oriented programs. In: 4th AOSD Modeling with UML Workshop, p. 18. UML (2003)
3. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston (2018)
4. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: Ernst, E. (ed.) ECOOP 2007—Object-Oriented Programming, pp. 600–624. Springer, Berlin (2007)
5. Cicchetti, A., Ruscio, D., Pierantonio, A.: Theory and Practice of Model Transformations: Second International Conference, ICMT 2009, Zurich, Switzerland, June 29–30, 2009. Proceedings, Springer, Berlin, Ch. Managing Dependent Changes in Coupled Evolution, pp. 35–51 (2009). https://doi.org/10.1007/978-3-642-02408-5_4
6. Di Ruscio, D., Iovino, L., Pierantonio, A.: What is needed for managing co-evolution in MDE? In: Proceedings of the 2nd International Workshop on Model Comparison in Practice, pp. 30–38. ACM (2011)
7. Anguel, F., Amirat, A., Bounour, N.: Using weaving models in metamodel and model co-evolution approach. In: 2014 6th International Conference on Computer Science and Information Technology (CSIT), pp. 142–147. IEEE (2014)
8. Wagelaar, D., Iovino, L., Di Ruscio, D., Pierantonio, A.: Translational semantics of a co-evolution specific language with the EMF transformation virtual machine. In: Hu, Z., de Lara, J. (eds.) Theory and Practice of Model Transformations, pp. 192–207. Springer, Berlin (2012)
9. Egea, M., Rusu, V.: Formal executable semantics for conformance in the MDE framework. Innov. Syst. Softw. Eng. **6**(1), 73–81 (2010)
10. Gruschko, B.: Towards synchronizing models with evolving metamodels. In: Proceedings of the International Workshop on Model-Driven Software Evolution held with the ECSMR (2007)
11. Di Ruscio, D., Iovino, L., Pierantonio, A.: A methodological approach for the coupled evolution of metamodels and ATL transformations. In: International Conference on Theory and Practice of Model Transformations, pp. 60–75. Springer (2013)
12. Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Traceability visualization in metamodel change impact detection. In: Proceedings of the Second Workshop on Graphical Modeling Language Development, pp. 51–62 (2013)
13. Pierantonio, A.: Supporting Users to Manage Breaking and Unresolvable Changes in Coupled Evolution
14. Rutle, A., Iovino, L., König, H., Diskin, Z.: Automatic transformation co-evolution using traceability models and graph transformation. In: European Conference on Modelling Foundations and Applications, pp. 80–96. Springer (2018)
15. Sánchez-Cuadrado, J., De Lara, J., Guerra, E.: Bottom-up metamodelling: an interactive approach. In: International Conference on Model Driven Engineering Languages and Systems, pp. 3–19. Springer (2012)
16. Di Ruscio, D., Iovino, L., Pierantonio, A.: Edelta: an approach for defining and applying reusable metamodel refactorings
17. Berg, H., Yu, I.C.: Generic Metamodel Refactoring with Automatic Detection of Applicability and Co-evolution of Artefacts, Research report. http://urn.nb.no/URN: NBN: no-35645. University of Oslo
18. Paige, R.F., Matragkas, N., Rose, L.M.: Evolving models in model-driven engineering: state-of-the-art and future challenges. J. Syst. Softw. **111**, 272–280 (2016)
19. Di Rocco, J., Di Ruscio, D., Iovino, L., Pierantonio, A.: Dealing with the coupled evolution of metamodels and model-to-text transformations. In: ME@ MoDELS, pp. 22–31 (2014)
20. Di Rocco, J., Di Ruscio, D., Pierantonio, A., Iovino, L.: Supporting users to manage breaking and unresolvable changes in coupled evolution. In: Proceedings of the Workshop on Domain-Specific Modeling, pp. 47–54 (2015)
21. Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A.: Edelta: an approach for defining and applying reusable metamodel refactorings. In: MODELS (Satellite Events), pp. 71–80 (2017)
22. Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A.: Quality-driven detection and resolution of metamodel smells. IEEE Access **7**, 16364–16376 (2019)
23. Gamboa, M.A., Syriani, E.: Using workflows to automate activities in MDE tools. In: International Conference on Model-Driven Engineering and Software Development, pp. 25–45. Springer (2016)
24. López-Fernández, J.J., Cuadrado, J.S., Guerra, E., De Lara, J.: Example-driven meta-model development. Softw. Syst. Model. **14**(4), 1323–1347 (2015)

25. Iovino, L., Rutle, A., Pierantonio, A., Di Rocco, J.: Query-based impact analysis of metamodel evolutions. In: 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 458–465. IEEE (2019)
26. Kusel, A., Etzlstorfer, J., Kapsammer, E., Retschitzegger, W., Schwinger, W., Schönböck, J.: Consistent co-evolution of models and transformations. In: 2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 116–125. IEEE (2015)
27. Di Ruscio, D., Iovino, L., Pierantonio, A.: Managing the coupled evolution of metamodels and textual concrete syntax specifications. In: 2013 39th Euromicro Conference on Software Engineering and Advanced Applications, pp. 114–121. IEEE (2013)
28. Herrmannsdoerfer, M., Vermolen, S., Wachsmuth, G.: An extensive catalog of operators for the coupled evolution of metamodels and models. In: SLE, Vol. 6563 of LNCS, pp. 163–182. Springer (2010)
29. Judson, S.R., Carver, D.L., France, R.: A metamodeling approach to model refactoring, OOPSLA. California, USA
30. Kessentini, W., Sahraoui, H., Wimmer, M.: Automated metamodel/model co-evolution using a multi-objective optimization approach. In: European Conference on Modelling Foundations and Applications, pp. 138–155. Springer (2016)
31. Koegel, M., Herrmannsdoerfer, M., Helming, J., Li, Y.: State-based vs. operation-based change tracking. In: Proceedings of MODELS, vol. 9 (2009)
32. Berg, H., Yu, I.C.: Generic metamodel refactoring with automatic detection of applicability and co-evolution of artefacts (2017)
33. Di Ruscio, D., Lämmel, R., Pierantonio, A.: Automated co-evolution of GMF editor models. In: International Conference on Software Language Engineering, pp. 143–162. Springer (2010)
34. Basciani, F., Di Ruscio, D., Iovino, L., Pierantonio, A.: Uncertainty management with extra-functional qualities in multi-artefact co-evolution. J. Object Technol. **20**(3), 1–15 (2021)
35. Di Ruscio, D., Iovino, L., Pierantonio, A.: Managing the coupled evolution of metamodels and textual concrete syntax specifications. In: 2013 39th Euromicro Conference on Software Engineering and Advanced Applications, pp. 114–121. IEEE (2013)
36. Vermolen, S. D., Wachsmuth, G., Visser, E.: Reconstructing complex metamodel evolution. In: International Conference on Software Language Engineering, pp. 201–221. Springer (2011)
37. Garcés, K., Jouault, F., Cointe, P., Bézivin, J.: Managing model adaptation by precise detection of metamodel changes. In: European Conference on Model Driven Architecture-Foundations and Applications, pp. 34–49. Springer (2009)
38. García, J., Diaz, O., Azanza, M.: Model transformation co-evolution: a semi-automatic approach. In: International Conference on Software Language Engineering, pp. 144–163. Springer (2012)
39. Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., Weiss, E.: EMF model refactoring based on graph transformation concepts. Electronic Communications of the EASST 3
40. Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.: Enhanced automation for managing model and metamodel inconsistency. In: 2009 IEEE/ACM International Conference on Automated Software Engineering, pp. 545–549. IEEE (2009)
41. Lombardi, T., Cortellessa, V., Pierantonio, A., Model, I.: Co-evolution of metamodel and generators: higher-order templating to the rescue. J. Object Technol. **20**(3), 1–14 (2021)
42. Demuth, A., Lopez-Herrejon, R.E., Egyed, A.: Automatically generating and adapting model constraints to support co-evolution of design models. In: 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pp. 302–305. IEEE (2012)
43. Lara, J.D., Guerra, E.: Refactoring multi-level models. ACM Trans. Softw. Eng. Methodol. TOSEM **27**(4), 1–56 (2018)
44. Pérez-Castillo, R., De Guzman, I.G.-R., Piattini, M.: Knowledge discovery metamodel-ISO/IEC 19506: a standard to modernize legacy systems. Comput. Stand. Interfaces **33**(6), 519–532 (2011)
45. Sghaier, O.B., Sahraoui, H., Famelis, M.: Metamodel refactoring using constraint solving: a quality-based perspective. In: 2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 797–806. IEEE (2021)
46. Lara, J.D., Guerra, E., Cuadrado, J.S.: When and how to use multilevel modelling. ACM Trans. Softw. Eng. Methodol. TOSEM **24**(2), 1–46 (2014)
47. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. CSUR **37**(4), 316–344 (2005)
48. Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A.: Detecting metamodel evolutions in repositories of model-driven projects. J. Object Technol. **19**(2), 14:1 (2020)
49. Iovino, L., Barriga, A., Rutle, A., Heldal, R.: Model repair with quality-based reinforcement learning. J. Object Technol. **19**(2), 17:1-17:21 (2020)
50. Barriga, A., Rutle, A., Heldal, R.: Automatic model repair using reinforcement learning. In: MODELS Workshops, vol. 2245, pp. 781–786 (2018)
51. Bettini, L., Di Ruscio, D., Iovino, L., Pierantonio, A.: Edelta 2.0: supporting live metamodel evolutions. In: ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, MODELS, pp. 1–10. ACM (2020)
52. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend, 2nd edn. Packt Publishing, Birmingham (2016)
53. Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., Hasselbring, W., von Massow, R.: Xbase: implementing domain-specific languages for Java. In: GPCE, pp. 112–121. ACM (2012)
54. Bettini, L., Ruscio, D.D., Iovino, L., Pierantonio, A.: Supporting safe metamodel evolution with edelta. Int. J. Softw. Tools Technol. Transf. **24**, 1–14 (2022)
55. Lloyd, J.W., Sonenberg, E.A., Topor, R.W.: Integrity constraint checking in stratified databases. J. Log. Program. **4**(4), 331–343 (1987)
56. García, J., Diaz, O., Azanza, M.: Model transformation co-evolution: a semi-automatic approach. In: Czarnecki, K., Hedin, G. (eds.) Software Language Engineering, pp. 144–163. Springer, Berlin (2013)
57. Di Ruscio, D., Iovino, L., Pierantonio, A.: Managing the coupled evolution of metamodels and textual concrete syntax specifications, pp. 114–121 (2013). https://doi.org/10.1109/SEAA.2013.22



Lorenzo Bettini is an Associate Professor in Computer Science at DISIA Dipartimento di Statistica, Informatica, Applicazioni 'Giuseppe Parenti', Università di Firenze, Italy, since February 2016. Previously, he was an Assistant Professor (Researcher) in Computer Science at Dipartimento di Informatica, Università di Torino, Italy. His research interests cover design, theory, and implementation of programming languages (in particular, Object-Oriented languages and Network-aware languages). More information is available at https://www.lorenzobettini.it/.

Davide Di Ruscio is an Associate Professor at the DISIM - University of L'Aquila. His main research interests are related to several aspects of Software Engineering, Open Source Software, and Model Driven Engineering (MDE) including domain-specific modeling languages, model transformation, model differencing, coupled evolution, and recommendation systems. He is in the editorial board of the International Journal on Software and Systems Modeling (SoSyM), of IEEE Software, of the Journal of Object Technology, and of the IET Software journal. More information is available at http://people.disim.univaq.it/diruscio.

Ludovico Iovino is currently an Assistant Professor at the GSSI - Gran Sasso Science Institute, L'Aquila-in the Computer Science scientific area. His interests include Model Driven Engineering (MDE), Model Transformations, Metamodel Evolution, code generation, and software quality evaluation. Currently, he is working on model-based artifacts and issues related to the metamodel evolution problem. He has been included in the organization and program committees of numerous conferences and workshops, and he is one of the organizers of the models and evolution workshop from 4 years. He is part of different academic projects related to Model Repositories, model migration tools, and Eclipse Plugins.

Alfonso Pierantonio is a full professor at the Università degli Studi dell'Aquila (Italy) and visiting professor at the Malardalens University (Sweden). His interests are in software engineering, model-driven engineering, and language engineering, with a focus on co-evolution techniques, consistency management, and bidirectionality. He has published more than 170 articles in scientific journals and conferences and has been on the organizing committee of several international conferences, including MoDELS and STAF. Alfonso is Editor-in-Chief of the Journal of Object Technology and on the editorial and advisory board of Software and System Modeling, and Science of Computer Programming. He has been PC Chair of ECMFA 2018 and General Chair of STAF 2015 and will be General Chair of MoDELS 2023; he is on the Steering Committee member of the ACM/IEEE MoDELS. He is a principal investigator of several research and industrial projects.