AUT Journal of Mathematics and Computing

AUT J. Math. Comput., 1(2) (2020) 179-216
DOI: 10.22060/ajmc.2020.17541.1027

A model transformation approach to perform refactoring on software architecture using refactoring patterns based on stakeholder requirements

Mohammad Tanhaei*[a]

[a]*Department of Engineering, Ilam University*

ABSTRACT: Software Architecture (SA) generally has a considerable influence on software quality attributes. Coordination of software architecture to the requirements of the stakeholders and avoiding common mistakes and faults in designing SA increases the chance of success of the project and satisfaction of the stakeholders. Making the wrong decisions at the architectural design phase usually proves very costly later on. Refactoring is a method which helps in detecting and avoiding complications, improving the internal characteristics of software, while keeping the external behavior intact. Various problems can undermine the architecture refactoring process. The existence of different requirements in different domains, the diversity of architecture description languages, and the difficulty of describing refactoring patterns lead to the difficulty of performing automatic and semi-automatic refactoring on the SA. In this study, we use model transformation as a way to overcome the above mentioned difficulties. In this regard, the first step is converting the SA to a pivot-model. Then, based on the refactoring patterns, the refactoring process is performed on the pivot-model. And finally, the pivot-model is converted back to the original (source) model. In this paper, the requirements of the stakeholders are taken into account in the refactoring process by modeling them as refactoring goals. These goals show the importance of the quality attributes in the project and the process of refactoring. The applicability of the framework is demonstrated using a case study.

Review History:
Received:15 December 2019
Revised:17 February 2020
Accepted:11 August 2020
Available Online:01 September 2020

Keywords:
Refactoring
Software architecture
Pattern
Model transformation

1. Introduction

Software architecture is one of the essential tools which helps in increasing the level of the software quality attributes. It plays a critical role in fulfilling the stakeholders quality requirements. On the downside, software architecture can have an adverse effect on the software quality attributes and decrease their level. The structure of the architecture, the type of the styles and patterns used to form it, the amount of detail used in describing the architecture, and tools used to analyze the architecture and ensuring its consistency are some of the factors that play a role in the success of using a software architecture. A primary way to avoid the negative side effects of the architecture is recognizing the common faults in designing SA and mitigate them using an appropriate approach.

Software architecture can be described using an Architecture Description Language (ADL). ADLs present a common understanding of the SA and enable users to perform some analyses on it. ADLs such as ACME [20], AADL [16], xADL [33], and MetaH [60] have been successfully used in the industry in the past decade [26]. Without using an ADL or some other well-defined language (that have a defined meta-model) to describe the SA we cannot expect to be able to perform analyses on the SA and find common faults in its design.

One of the proper approaches to recognize and resolve design faults in the SA is software refactoring. Software refactoring was introduced by Opdyke in 1992 [47]. The aim of refactoring is to improve the internal quality of

*Corresponding author.
E-mail addresses: m.tanhaei@ilam.ac.ir

software without changing the external behavior and the functionality of software [17]. In the past years, several works have been done to extending the refactoring concept on software artifacts other than source code. These works extend refactoring to a higher level of abstraction such as design, architecture, and software requirements. While the focus of refactoring at the code-level is on improving software maintainability, refactoring at the higher levels also has to consider other quality attributes [67].

Improving some quality attributes at code level is rather difficult, requiring large scale changes to be made to software. For example, improving the performance of a software system cannot be done simply by refactoring the code. Improving performance sometimes needs large changes in the SA, and the application of some performance enhancing styles. Performance, security, and other quality attributes can be affected positively or negatively while using various architectural patterns.

1.1. Motivations for having a tool

In this section, we briefly discuss the need for a refactoring tool at the software architecture level. In manual performing of the SA refactoring, one may face these difficulties:

M1. Finding refactoring opportunities is difficult: The conditions of performing some of the refactoring patterns are complicated. One needs to check several elements of the SA and their relation to each other in order to find a refactoring opportunity in it.

M2. Performing refactoring patterns is time-consuming and error-prone: Finding a refactoring opportunity in the SA is not the end of refactoring. One may need to change several elements of the architecture and their relation for carrying out the refactoring pattern. One of the simplest refactoring patterns on the SA is *rename*. Consider renaming an element which has n relations to other elements of the architecture: One needs to check and modify all these n relations to perform the rename refactoring. It is also important to note that making changes to the SA elements and their relations without using a tool is error-prone (similar to other human-related works).

M3. Checking the correctness of the refactoring is time-consuming: After performing the refactoring, one should test the functionalities of the SA to ensure that the set of the functionalities of the SA before and after performing SA refactoring is not changed. This action may be extremely time-consuming. For example, in merging two elements of an SA, one needs to check all members of the two and the relation of them to other elements of that SA.

M4. Knowledge of the humans is limited: Finding a person who knows about all refactoring patterns which are applicable on an SA in detail is hard. Using a tool we can aggregate knowledge of the community and remove the dependency on people. Providing a tool to reuse the refactoring patterns can reduce the amount of effort needed for specifying them and could improve the synergism in describing them.

M5. Calculating the trade-off among refactoring patterns is challenging: Consider the situation that multiple refactoring patterns can be applied to a software architecture simultaneously. In this situation, one should calculate the trade-off among refactoring patterns and choose the refactoring pattern that satisfies the stakeholders requirements to the highest possible degree. Calculating the trade-off among refactoring patterns without using a tool is challenging and time-consuming.

1.2. Problems

The refactoring process at architecture-level as the highest level of abstraction in the software development process faces several challenges. We ought to provide appropriate solutions to these challenges to be able to design a framework for supporting SA refactoring.

Ch1. The first challenge is the lack of appropriate tools to aid in performing refactoring. Refactoring tools can assist in finding refactoring opportunities in the software architecture and help in avoiding common mistakes in designing architecture and improve the overall quality of the software architecture. In order to perform refactoring on an ADL, one needs to use appropriate tools designed for that ADL. However, the existence of a plethora of different ADLs, arising from the diversity of domains and requirements, limits the tools and their applications. Defining separate refactoring tools for each ADL and defining the corresponding refactoring patterns is not an easy task.

Ch2. The second challenge in performing SA refactoring is the difficulty of cataloging refactoring patterns, bad smells, and architectural styles.

Ch3. The way that the refactoring goals are specified and used in the refactoring activities present another challenge in performing SA refactoring. The user has to specify the goals of the refactoring to perform software architecture refactoring. Refactoring goals show what the main aim of performing refactoring is.

1.3. Solutions

The solutions provided by our framework to overcome the mentioned challenges are as follows:

S1. Model Driven Engineering [36] (MDE) approaches have been used successfully to solve the challenges which are similar to the first challenge (Ch1). We use the model transformation tools which are used in the MDE approach to produce a pivot-model for each ADL. After transforming the ADL to a pivot-model, the refactoring patterns can be applied without worrying about the syntax of a particular ADL..

By transforming the ADL to a pivot-model, the refactoring patterns defined on that pivot-model can be reused. Reusing the refactoring patterns can save the time needed to define them and increase the maturity of the defined patterns by reusing and reassessing them repeatedly. Finally, the refactored pivot-model is converted back to the original form (source model). The transformation needs the semantics of the model, which is in turn provided by meta-models. Using model transformation languages in our framework facilitates defining new refactoring rules by the users.

S2. Several works have been done on describing patterns. One of the approaches uses formal description to describe the patterns. Another approach uses OCL to describe patterns. In this paper, we introduce a way to describe refactoring patterns, which can be reused in several projects and in different refactoring processes. To face the second challenge (Ch2), we use a UML profile to describe the different properties of an SA refactoring pattern. The relationship between the refactoring patterns and the quality attributes will be considered in designing this UML profile.

S3. To solve the Ch3 challenge, we develop an approach, which uses quality attributes, to describe the refactoring goals. The importance of each quality attribute is described using a positive integer. One of the advantages of our framework over similar frameworks is using refactoring goals in the refactoring process. In our framework, stakeholder(s) of the project specify the refactoring goals using the AHP method.

1.4. Paper Structure

The structure of the remainder of this article is as follows. After presenting some essential background in Section 2. We introduce the framework for performing refactoring on software architectures in Section 3. Section 4 evaluates the applicability of the framework in an example domain. Threats to validity of the result of the proposed framework are assessed in Section 5. In Section 6, the idea of the paper, weaknesses, challenges, and opportunities are discussed. We survey related work in Section 7. The paper concludes in Section 8 with a summary and a discussion of possible future work.

2. Background

2.1. Model Driven Engineering

Model Driven Engineering is an approach used to deal with the complexity of different domains and their optimum description [61]. This approach has been developed in response to the inability of the third-generation languages to specify the plethora of needs present in different domains. MDE uses Domain-Specific Modeling Languages (DSMLs), as well as Generators and Transformation tools [32].

Transformation tools and generators have the ability to perform a wide range of analyses on software artifacts from code-level to architecture level to documents. Using these tools increases the compatibility between different software artifacts and reduces possible faults.

2.2. Model Transformation

A model is an abstraction of the real world. Each model has a distinctive approach to abstract the world. It is possible that the same real world object is portrayed differently by different modeling languages. By describing their semantics, the meta-model gives us a unique understanding of the objects.

Model transformation is the description of model to model (M2M) conversion in a specific domain. In this approach, the semantics of the model comes from the model's meta-model. Mens *et al.* [41] introduce two types of transformation: Exogenous and Endogenous. The differences between the two types lie in the relation between source and target meta-models. The meta-models of the source and the target in the Endogenous transformation are the same, whereas in the Exogenous ones they are different. In this research, transforming an ADL to a UML profile is exogenous, while SA refactoring is an endogenous transformation.

2.3. Formal Description of Patterns

For automating the process of SA refactoring one should specify the patterns which should be checked on the SA. Different approaches have been employed to describe architectural and design patterns. Formal specification of patterns is used in [38, 37]. Mikkonen [42] used Temporal Logic of Actions to propose a pattern specification notation called DisCo. LePus, a notation for specifying pattern properties, was introduced by Eden [15]. Dietrich *et al.* present a formal pattern description language in [11]. This language can be used to describe patterns at higher-level artifacts. A formal method for defining design patterns is developed by Lano *et al.* [37].

3. A Framework for Software Architecture Refactoring based on the Model Transformation

In this section, we introduce our approach to perform refactoring on the software architecture using model transformation. In Section 3.1, the general methodology of performing refactoring based on model transformation is presented. To carry out our methodology, we implement several components.

The type of activities in our framework are divided into three major categories:

- *Regular User level activities*: Defining the refactoring goals and architecture instances are the two activities done at this level. The activities at this level need knowledge about defining architecture with the specific ADL we want to perform refactoring on its instance and knowledge about how to model refactoring goals in our framework.

- *Expert level activities*: Definition of the Domain Specific Languages (DSL) pivot-model, the transformation rules, and the SA refactoring rules are the main activities in this level. These activities are performed just once in refactoring a specific ADL. Once the pivot-model for describing an ADL is designed, it can be reused repeatedly in many refactorings performed on that ADL. This fact is the same about the transformation rules and the SA refactoring rules. Expert knowledge about the pivot-model (e.g. a UML profile), the transformation language (e.g. ATLAS Transformation Language (ATL) [29]), and the constraint language (e.g. OCL) is a prerequisite for performing activities at this level.

- *Tool level activities*: These activities are common to all SA refactorings performed using our framework. Performing refactoring and decision making are the two main activities in this category. Our algorithm for making decisions and performing refactoring can be reused on any transformation once implemented. As a result, the user of the tools (whether an expert or a regular user) need not deal with implementing or providing these activities in our framework.

3.1. The General Methodology of SA Refactoring

One of the biggest difficulties in performing SA refactoring is the lack of appropriate analysis tools (See Section 1.1 **M1-5**). Other difficulties include the extreme dependency of the refactoring patterns on the ADL notation (**Ch1**), and the ad hoc nature of SA refactoring (**Ch2**). These difficulties cause the hardness of performing SA refactoring, and mean the refactoring process is mostly a human activity, with few automated tools available to aid in the process.

Within this framework, our main goal is to eliminate dependence on a specific language and create a tool to perform SA refactoring on various domains (**S1**). Accordingly, ADLs are transformed to a pivot-model (e.g. a UML profile) and various refactoring patterns are described on that model. The characteristics of the ADLs such as the type of elements and the type of connection between elements in a specific domain are very similar, and the differences in the languages lie mainly in the amount of the detail they can describe. Using a pivot-model which describes most characteristics of a domain means the majority of the ADLs on that domain can be transformed to that pivot-model.

Figure 1 shows an overview of the SA refactoring approach proposed in this framework. To perform SA refactoring, the ADL code is transformed to an appropriate pivot-model. In this paper, we use transformation languages to transform the source model to the designed pivot-model. To do the transformation, one requires access to the meta-model of the source and the pivot-models. Meta-model is a model that shows the arrangement of the elements of the model and indicates the interactions between them. For transforming a model to another model using a transformation language, the user should specify the mapping policy by transformation rules.

The refactoring process begins when one does the transformation of the source model to the pivot-model. A pivot-model, refactoring goals, and refactoring patterns constitute the parameters based on which the refactoring process works. Refactoring goals specify the aim of SA refactoring, and direct the process of selecting applicable patterns. In situations where multiple refactoring patterns are applicable, it is the refactoring goals that ultimately determine which pattern leads to what level of satisfaction, and thence how to prioritize the patterns to apply. Another input to the refactoring process is the refactoring patterns. Refactoring patterns determine applicable

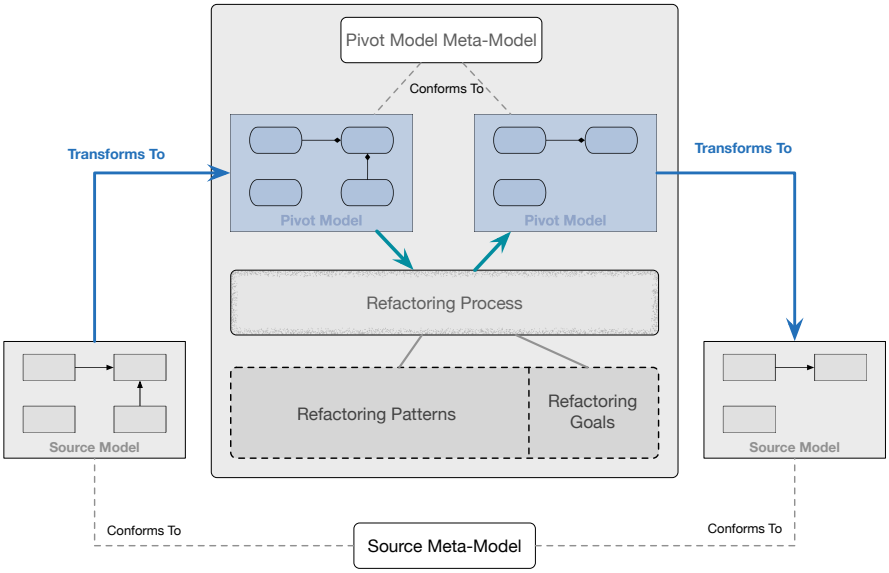

Figure 1: The general methodology of SA refactoring proposed in this paper.

patterns in a specific domain. These patterns describe the source conditions and specify the target shape and post-conditions which should be met. The refactoring patterns delineate the amount of improvement (or the lowering) in the quality and sub-quality attributes of the software. All the pattern constraints can be described using a constraint language (e.g. OCL).

The refactoring process starts with an instance of the source model, and then the refactoring patterns which can be applied in regard to refactoring goals on this instance are performed on it. The first stage of the refactoring process is prioritizing refactoring patterns according to the objectives of SA refactoring. The prioritizing algorithm is illustrated in Algorithm 3. After prioritizing the patterns, the matched patterns with a higher priority are performed. At the end of the refactoring process, the pivot-model is transformed to the original form using transformation rules.

3.2. Expert Level Components and Activities

3.2.1. A pivot-model for defining DSL

ADLs generally fall into two categories: General-Domain and Domain-Specific. One of the approaches for analyzing ADLs is to transform them into a pivoting model and then check the refactoring conditions on that model. General-Domain ADLs can be described using a pivot-model like the one provided in [25]. Describing Domain-Specific ADLs requires specific pivot-models, which are designed to support special needs of those domains. For this kind of ADLs, one can use predefined pivot-models (e.g. UML profiles) if they exist.

Before we introduce our approach towards creating a pivot-model for supporting a DSL, it is appropriate to take a look at the key elements in the meta-models:

1. *Fundamental Language Constructs*: The meta-model of a domain-specific modeling language contains the essential concepts of that domain. For example, a DSML for modeling real-time applications may have processors, queues, tasks, etc.
2. *Relationships*: The set of valid relationships that exist between language elements in the meta-model. In our example, a processor may work on various tasks. (a relation between the processor and the relevant tasks exists)
3. *Constraints*: The constraints that exist on the combination of the language elements in the meta-model. In our example, a processor can only work on one task at a given time.

Each element of the DSL meta-model should be considered in designing the pivot-model to map DSL elements into. In what follows, we suggest a guideline for designing a pivot-model appropriate for supporting a specific DSL language based on the work of Selic [56]. This guideline uses the DSL meta-model to design an appropriate pivot-model.

1. *Selection of base meta-classes*: For supporting each type the essential concepts of the DSL which are present in the meta-model, we should select a proper meta-class. The semantics of the selected meta-class should be close to the semantics of the domain concept. This is important because the people who know pivot-model semantic, expect a particular behavior from a specific meta-class and its inherited classes. Using the elements

whose semantics are closest to the semantics of the essential concepts in designing a pivot-model facilitates its understandability.

2. *Adding/Refining attributes to/from the base meta-class*: Adding/Refining attributes to/from the base meta-class are ways to specialize the meta-class for use in a specific domain. The semantics of the DSL are transferred to the pivot-model by adding or refining attributes to/from the selected base meta-class. An investigation on the concept of the DSL would reveal the attributes needed to support it by a meta-class in the pivot-model metamodel. For example, a processor may have clock speed, isActive, etc. attributes in a DSL designed to model real-time computation.
3. *Checking the constraints*: All the constraints that can be applied on the base meta-class should be checked to detect any conflicts.
4. *Checking the associations*: The associations among the meta-classes selected to support DSL should be investigated and verified that these associations do not have conflicts. A large number of conflicts can be avoided by setting lower multiplicity bound to zero. If conflict cannot be avoided, the type of the meta-class selected to support a concept in the DSL should be changed to another meta-class type appropriate to support it.

Implementation. We used UML profiles to implement the pivot-model for supporting DSL models in our framework. UML structure consists of different abstraction levels [58]. The M0 level consists of the actual instances of the system. Various system models constitute the M1 level. The instances of the M1 models are in M0 level. The M2 level contains elements to describe models in level M1 (Meta-models). Finally, the M3 level consists of models to describe models in the M2 level. This level is called the meta-meta-model level. The Ecore [59] modeling language is in the M3 level.

Because not all elements of other models can easily be transformed to UML elements, to transform a model to UML model, one usually needs to provide a specific profile. The UML infrastructure contains UML profile extension mechanisms [58]. UML profiles provide compatibility between UML meta-models and platforms, domains, business objects, and software processes. UML profiles are an appropriate tool for describing DSL. Software architecture in different domains can be modeled using this UML feature [21, 1]. Compared to other languages and tools, the advantages of UML profiles include the availability of OCL constraints and the vast number of available tools. A sample of designing UML profile is demonstrated in the case study in Section 4.2.2.

3.2.2. Refactoring Pattern Specifier

At minimum, every refactoring pattern has these parts:

- Context: Every refactoring pattern is applicable only in a specific context. The pattern applicable in the context of real-time software systems may not be suitable to be used in the context of a data-intensive software system.
- Refactoring Conditions (Problem): The refactoring conditions which must be matched on the model in order to perform refactoring.
- Refactoring Action (Solution): The actions done on the software artifacts such as architecture to perform the SA refactoring.
- Refactoring Post Conditions: Post conditions are the conditions that must be met to ensure that the refactoring is behavior preservative. Having post conditions guarantees the preservation of the external behavior of the software after performing refactoring by examining the Original and Refactored models and comparing the set of the functionalities, relations, and configurations of the changed components in the Original and Refactored models.
- Refactoring Goals: Performing each of the SA refactoring patterns affects some of the quality attributes of the software systems. The effects lead to an increase or a decrease in the amount of the corresponding quality attributes.

In our refactoring approach, every ADL is transformed to a pivot-model and the conditions of the SA refactoring pattern in the context of that ADL are checked on that pivot-model. If the conditions are matched on the pivot-model, the refactoring activities are performed on it and refactoring postconditions are checked to verify that the behavior has been preserved after performing refactoring.

A refactoring rule is defined by five elements: its name, a guard which defines when the rule can be applied, a set of parameters, a body that implements the effect of the rule, and a post condition rule which checks the effect of the rule and ensures behavior preservation of the rule. Because we use the ATL [29] language for implementing our framework, and it supports OCL constraints in its syntax, we can use OCL to write refactoring conditions, actions, and post conditions. We use ATL rules to write the refactoring conditions and refactoring actions like the code snippet below:

```
rule RefactoringName{
  from [refactoring condition in ATL & OCL]
  to [refactoring action in ATL & OCL]
  do { [refactoring action in ATL & OCL (imperative)]}
}
```

We want to show a refactoring on a simplified version of the class diagram here. In order to perform refactoring in our framework, we need the meta-model of the model we want to do refactoring on. We introduce the meta-model of a simple version of the class diagram in Figure 2. We include the essential and simple part of the class diagram in our meta-model. This meta-model supports simple classes and their relations to each other. In this figure (Figure 2), each *Class* contains some *Operations* and *Attributes*. Every *Operation* has some *Parameters*. Every *Parameter* has only one *GenericType*. Every *GenericType* relates to *DataType* or *Class* class. Every *Attribute* has only one *GenericType*. Every *Reference* can relate to one additional *References* class. And finally every *Class* can have some *References*.

One well-known refactoring is folding. Figure 3 shows a graphical definition of folding refactoring. In this figure, two methods with the same name and body in two classes are moved to the same location. Here we show a refactoring rule for performing folding refactoring on the models which conforms to the meta-model we introduced in Figure 2.

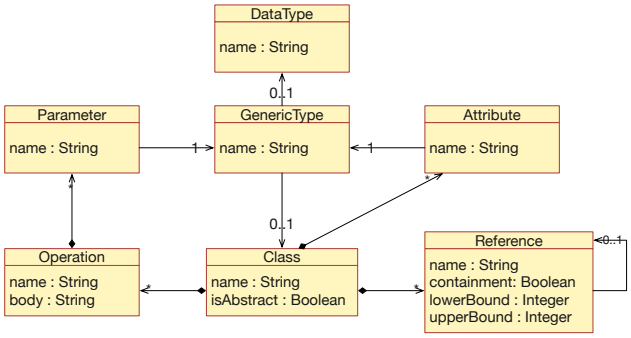

Figure 2: Simple Class Diagram Meta-Model

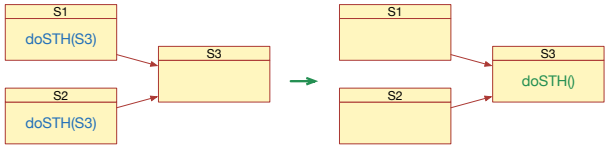

Figure 3: Folding refactoring between three classes in UML

The first step in defining a refactoring rule is specifying refactoring conditions (guards). The condition of refactoring in this sample is:

- Methods body are the same.
- Methods belong to different classes.

We use OCL and ATL to write the above conditions as follows:

```
rule FoldingRule{
  from ops : SMC!Operation (
    SMC!Operation.allInstances()
     ->select(p1, p2 | p1.body = p2.body and p1.name = p2.name and p1.class != p2.class)
   )
}
```

In order to perform refactoring we need to access and manipulate the parameter of the operation we used. And for finding *S3* (The target class) we need access to the related GenericType. So we modify our rule as follows:

```
rule FoldingRule{
  from  ops : SMC!Operation (SMC!Operation.allInstances()
     ->select(p1, p2 | p1.body = p2.body and p1.class != p2.class)),
   param : SMC!Parameter (ops->exists(o | param.operation.name = o.name),
   type : SMC!GenericType (param->exists(p | type.name = p.type.name)
}
```

Now we can perform refactoring on the selected *Operation* and *Parameters*. We want to drop two similar operations, their parameters (including related types) from class *S1* and *S2*, and add new operations to class *S3*. We utilize *findClass* helpers to do this job. The *findClass* helper finds the target class to which we want to move the method (this is the *S3* class in our example). In the code snippet below we first add a new operation to class *S3* and then drop all selected entities from *Operations*, *Parameter* and *GenericType* class.

```
1 helper context SMC!GenericType def : findClass : SMC!GenericType = self->iterate(p|
2   if (p.class != null) then p.class endif);
3
4 rule FoldingRule{
5   ...
6   to drop
7   do {
8       st = ops.getAppliedStereotypes()->first();
9       ops.setValue(st, "class", type.findClass);
10      ops.setValue(st, "param", null);
11      ops.applyStereotype(st);}
12 }
```

Refactoring patterns can exist in various forms. Architectural styles, design patterns, and bad smells can be a source for defining refactoring patterns. However, defining an architectural style as a pattern is a challenging and complicated undertaking. We describe a concrete example of defining the refactoring rules in Section 4.2.5

Implementation. There are many methods for describing SA refactoring patterns. These methods are surveyed in Section 7 of this paper. In our framework, we use a UML profile to describe refactoring patterns.

The context profile for describing refactoring patterns is shown in Figure 4. The following provides a brief explanation for each part of the profile:

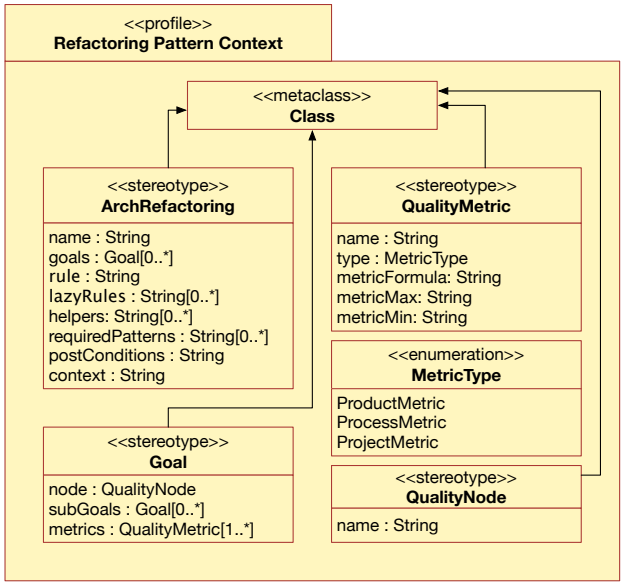

Figure 4: Context Profile of Software Architecture Refactoring Patterns

- ArchRefactoring Stereotype: This stereotype describes constraints on transforming from a source model to a target model. Each refactoring pattern affects a different range of quality attributes. The *goals* variable in ArchRefactoring reflects the effect of the pattern on the quality attributes. *Rules* specify the ATL rules like the code snippet introduced before. *Lazy Rules* are the helper rules needed by the main refactoring rule. Lazy rules in ATL are the rules which can be used declaratively (they need to be called). *Helpers* variable contains the helpers which are used by the current pattern rules. The helpers are the peripheral functions used by main refactoring rule. *Required patterns* are the patterns that the current pattern uses (calls) during refactoring. The post conditions which should be met in performing a refactoring rule can be specified by ATL and OCL, and are stored in the *postCondition* variable. After a refactoring pattern is applied, our framework checks its post conditions. Our framework keeps a copy of the architecture before performing a refactoring and rolls it back if the post conditions of the refactoring pattern are not met. *PostCondition* can use the *Old* or *New* model to access the model before and after refactoring. An example of using *postCondition* is described in

the case study section. The *context* variable specifies the UML profile (the middle model to which the source model is transformed) which the current refactoring pattern belongs to.

- Goal Stereotype: This stereotype describes the effect of the refactoring pattern on the quality and sub-quality attributes. *Node* specifies the related quality or sub-quality attributes. *Subgoals* specify sub-quality attributes of the current quality or sub-quality attribute. *Metrics* measures the effect of the pattern on the current quality or sub-quality attribute.

- QualityMetric Stereotype: This stereotype maintains the metric formula and is used to measure quality and sub-quality attributes. *MetricFormula* specifies the formula needed to compute the metric value in the refactoring context. Refactoring framework evaluates the *metricFormula* string on the UML profile which the source model is transformed to. The result of the formula evaluation on the UML profile is an unnormalized value (FormulaVal).

  Formula 1 demonstrates the normalization formula. The value of *metricMax* and *metricMin* in Formula 1 are determined experimentally. One can find the *metricMax* and *metricMin* values by playing with the parameter(s) of the metricFormula string.

$$NormalMetric = 6 * \frac{FormulaVal - metricMin}{metricMax - metricMin} - 3 \tag{1}$$

- MetricType Enumeration: This enumeration type specifies the type of the metric. A metric has one of the following three types: Product Metric, Process Metric, or Project Metric [30].

- QualityNode Stereotype: This stereotype defines the name of the quality and sub-quality attributes. This name is used for discovering the QualityAttribute class in the search and reasoning algorithms.

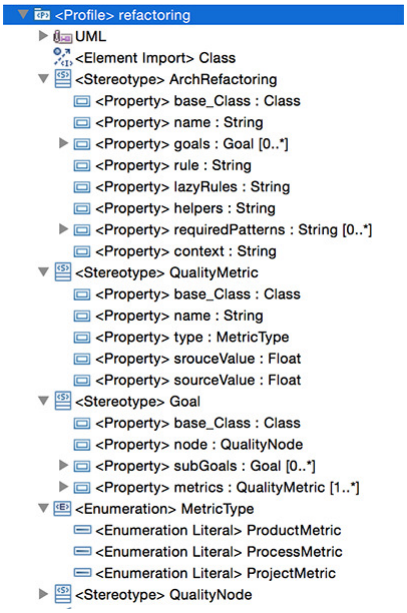

Figure 5: Refactoring Pattern Profile in the EMF [59]

We provide several UML profiles in this paper. The users can apply these profiles, and import the profile instances to the refactoring process. Figure 5 shows refactoring pattern profile in the Eclipse Modeling Framework (EMF) [59]. Applying UML profile can be done by using *UML Editor* in EMF or by using *org.eclipse.uml2* Java library. We utilized this library in our tool.

3.2.3. Models Transformer

Model transformation can be done using general-purpose (e.g., Java) or M2M (e.g., QVT [23], ATL [29] and Xtend [46]) languages. Transforming a model to another model in general-purpose languages requires a large amount of code to be written. The hardness of transforming models using general-purpose languages is the existence philosophy of M2M languages. These languages facilitate the transformation process.

Implementation. In this framework, we used ATL transformation languages [23] to do the transformation. ATL is an element of the Eclipse M2M sub-project of the EMF [59]. ATL enables the user to transform a model which conforms to a meta-model to another model. According to [8] ATL is the fastest language for transforming models to each other. In this language, full control on the transformation process from source to target exists. Using the ATL language has some advantages for our refactoring framework. One of the advantages of ATL is the ability to use OCL in the transformation rules. The other advantage of ATL is the existence of a Java library which can be used in standalone programs. The code snippets below shows a simple transformation from ACME ADL to the related UML profile.

```
module ACME2UML;
create OUT : ACMEPROFILE from IN : ACME;

rule Element2EelementProfile{
  from  m : ACME!Element
  to  a : ACMEPROFILE!Element (
    a.name <- m.name
    a.property <- prop
    a.representations <- reps
    )
}
```

In order to perform model transformation, one needs the semantics of the source and target models. For our purpose, the semantics of a model can be defined by using meta-model languages. Using the meta-model of particular models, elements of the models can be transformed into one another. One of the meta-model description models widely used in the open-source world is the Ecore model in the EMF [59]. Ecore is a meta-meta-model which can be used to describe the meta-model of a specific model in Eclipse. In this research, we used the Ecore meta-meta-model to describe the source meta-model.

For transforming the source model to target model we also need a mapping rule. The mapping rules describe the way each element of the source model should be present in the target model. The source model is transformed to the target model using these mapping rules. In the case study of this research, we present a transformation from ACME ADL [20] to a UML profile.

3.3. Regular User Level Components and Activities

3.3.1. Refactoring Goals Specifier

Refactoring at the architecture level usually seeks various goals. Unlike code refactoring whose main purpose is focused on increasing maintainability of software, architecture refactoring can target a wider range of quality attributes. Therefore, in performing refactoring on software architecture, one might have multiple goals in mind [64]. Our ultimate purpose is to attain the refactoring goals to the maximum extent possible.

In order to specify the refactoring goals for a project, we need a categorization of the quality attributes. There are various software quality attribute classification models. Some of the more stablished are Dromey [13], Boehm [7], McCall [40], and ISO 9126 [27]. We use the ISO 9126 [27] quality attributes classification standard in this framework. In the ISO 9126 standard, quality attributes are divided into 6 categories. Each quality attribute consists of some sub-quality attributes. Figure 6 shows the ISO 9126 classification of the quality attributes. In the ISO 9126-2 [27] standard some metrics are defined for measuring the value of the quality and sub-quality attributes which are used in the ISO 9126 quality model.

However, our framework and algorithms for refactoring on the SA are developed independently of the quality models. As a result, other quality models can be also used in our framework.

Each quality attribute is related to some sub-quality attributes. The importance of each quality and sub-quality attribute differs in various projects. In a military project, security, reliability, and performance have the highest importance while in a Customer Relation Management (CRM) related project, the focus is on usability and reliability. Due to this fact, some refactoring patterns can be ideal for some kinds of projects, but be costly and wasteful for other kinds of projects. In other words, not every refactoring pattern is suitable for every project.

One has to evaluate the importance of quality and sub-quality attributes in a given project, and then decide on the subsequent course of action.

In this section, we describe a refactoring goals specifier which can be used to define refactoring goals for a software project. The refactoring goals specifier gives us the ability to describe the importance of quality and sub-quality attributes related to a software project. It relates each quality attribute to some sub-quality attributes with a specific importance factor. The related quality attributes can form a graph. In the formed graph, the weight of each edge shows the importance of each quality and sub-quality attributes of the project. The nodes in the formed graph are quality and sub-quality attributes. A sub-quality node can have only one parent in that graph (the graph is a tree). Figure 7 shows a refactoring goals graph for a CRM-Related project.

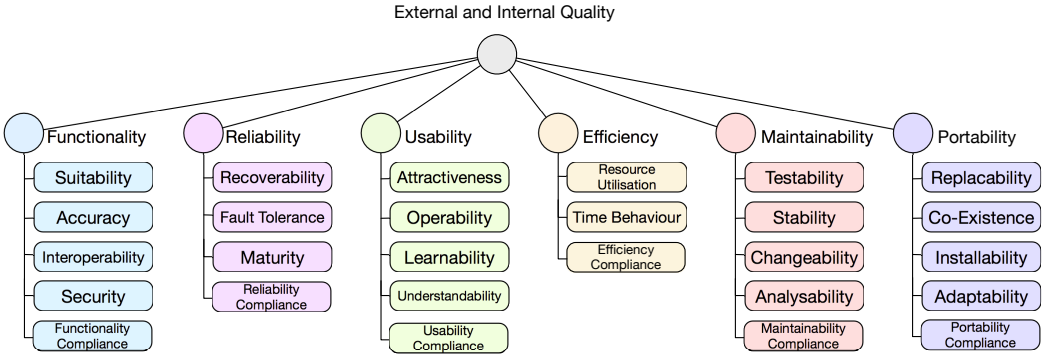

Figure 6: ISO 9126 [27] classification of quality attributes. The quality attributes are placed into 6 main quality and 27 sub-quality attributes.

Refactoring goals are a collection of quality and sub-quality attributes related to a project. Each quality attribute has an importance factor ranging from 0 to 10. In a sample project, it is possible that a quality or sub-quality attribute has a higher importance for the stakeholders. In this situation, the stakeholders can assign a higher value to the *importanceFactor* of the corresponding quality or sub-quality attribute. In our framework, the value of *importanceFactor* is used to decide about the proper refactoring patterns in the refactoring process.

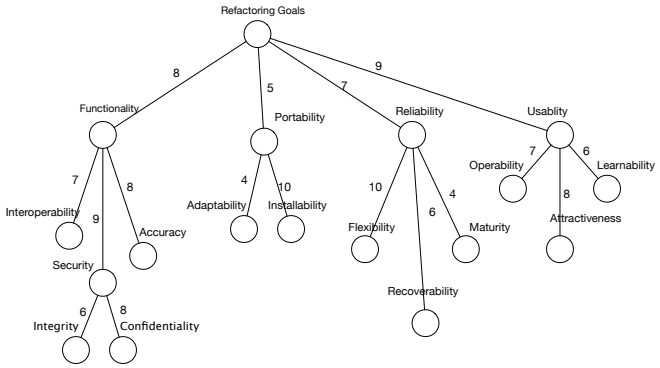

Figure 7: Refactoring Goals Graph For a CRM Related Project

The *importanceFactor* of each quality and sub-quality attribute can be assigned using different approaches. One of the appropriate approaches to calculate the *importanceFactor* is to use the Analytic Hierarchy Process (AHP) method [55]. The AHP approach is used mainly to support decision making in complex decision problems. However, we utilized the method used in AHP to calculate priorities, and leave out the decision making part of the AHP. The AHP approach (for use in our problem) takes three steps to calculate the importance factor of each element in a graph.

1. Dividing a problem to some sub-problems (and to sub-sub-problems), and forming a hierarchy.
2. Finding the priority of the (sub-)sub-problems to each other (pairwise comparisons).
3. Forming a priority matrix and computing the overall priority of each (sub-)sub-problem.

The first step in using the AHP method is to divide the problem into some sub-problems and form a tree of the problem and sub-problems. There are several well-known quality models in the context of quality attributes (mentioned before). We used the ISO 9126 [27] standard to categorize the quality attributes in some (sub-)sub-quality attributes.

The second step in using the AHP method is to find the priorities of each quality and sub-quality attribute relative to each other. The pairwise comparison is done between siblings in the hierarchy. For example in Figure 7 the stakeholder(s) should provide pairwise comparison between Functionality, Portability, Reliability, and Usability. The stakeholder(s) of the project can assign a score between 1-9 in comparing two (sub-)qualities. In the situation in which stakeholders have different views on the quality attributes and the pairwise scores are different, a weighted average of the pairwise comparison scores between quality attributes can be used. The scores of the stakeholders with higher importance will take a higher weight in computing the average. Table 1 shows the relative scores used

Table 1: Relative scores between j and k in AHP [55]

Score	Interpretation
1	Two elements are **equally** important
3	j is **slightly more** important than k
5	j is **more** important than k
7	j is **strongly more** important than k
9	j is **absolutely more** important than k
2,4,6,8	Middle scores

in the AHP approach [55]. The output of the AHP calculation does not depend on the range of the comparison table [55]. As a result, users can use any other relative scores such as a simple range of 1 to 3 to compare the quality attributes. Assigning a score to the pairwise comparison between two quality or sub-quality attributes is not difficult for the stakeholders.

The last step in using AHP to calculate the priority of the quality and sub-qualities is to form a matrix from pairwise comparison scores. This step can be done using tools such as PriEsT [57]. The result of calculation is priority scores for each quality or sub-quality attribute. These scores are floating point numbers between 0 and 1. The stakeholders can use these numbers in forming refactoring goals or simply multiply all the values by an appropriate number and use the modified numbers in forming refactoring goals.

Implementation. The graphical presentation of the refactoring goals is not appropriate for using in our framework. We designed a UML profile to capture this information from the user (Figure 8). The user should apply this UML profile for importing the refactoring goals to our framework.

We briefly explain each part of this profile:

- RefactoringGoal Stereotype: This stereotype specifies the most important quality attributes in the project. Improving these quality attributes provides tangible benefits to the stakeholders.

- QualityAttribute Stereotype: This stereotype specifies the importance of each quality and sub-quality attribute of the project. The *node* variable specifies the quality or sub-quality we focus on. The *importanceFactor* has a value ranging from 0 to 10. This value is assigned by project stakeholders. Project importance, project type, the external forces, and team experience usually determine the value of *importanceFactor*. Each quality attribute can have some sub-quality attribute, which is specified by the *subQualities* variable.

- QualityNode Stereotype: This stereotype has already been described in Section 3.2.2.

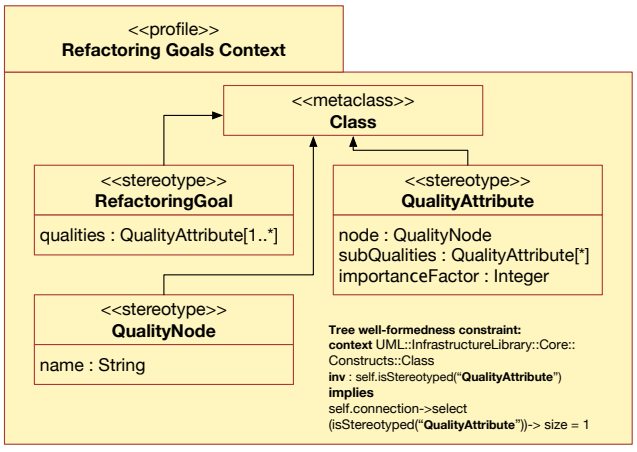

Figure 8: Context Profile of Refactoring Goals

3.4. Tool Level Components and Activities

3.4.1. Weight Calculation Component

After defining all applicable refactoring patterns on the pivot-model, decision has to be made about which are the most suitable patterns to apply. The most appropriate patterns should be selected in regard to the refactoring goals. This is because, as noted before, one pattern may be appropriate for some projects, but be inappropriate for some other projects. In this section, we provide algorithms for computing the propriety of each refactoring pattern in regard to the refactoring goals.

Each refactoring pattern changes the value of one or more quality attributes. The amount of *NormalMetric* before and after performing refactoring shows the amount of this impact. The larger the difference is, the more positive the impact on the quality attribute is. A negative difference indicates a negative impact on the quality attribute. The difference ranges from -6 to +6 from the most negative to the most positive impact on the quality attribute.

We use *OveralQFactor* for computing the overall importance of a quality attribute in the SA refactoring. Algorithm 1 shows the procedure used for computing *OveralQFactor*. The computation of *OveralQFactor* is based on the importance value which the stakeholders have characterized in the refactoring goals.

In Algorithm 1, we aim to calculate the overall quality factor of a node Q in the graph G. At the beginning of the algorithm, the value of this factor is zero. We look for a parent of the Q node in graph G in Line 1. In the next line of code (Line 1) we grab the QualityAttribute stereotype instance (See QualityAttribute in Figure 8) related to the Q node. The QQ variable contains a copy of the QualityAttribute instance. It contains the *importanceFactor*, *node*, and *subQualities* of the Q quality. If the quality node has no parent in the refactoring goals graph G we have reached the root node, and the amount of overall quality factors of the root node equals one (Line 1). Otherwise, we continue our algorithm in Line 1 by setting the child_factor to zero. The code in Line 1 iterates for each child of the parent of the node Q (siblings of the Q). We sum the importantFactor of the siblings (including Q itself) of the Q quality and put them in child_factor. In line 1, by dividing the amount of the Q importanceFactor by the child_factor we attain the relative importance of the quality attribute Q to its siblings. However, for computing the overall importance of the Q attribute, we need to calculate the relative importance of its parent and multiply it by the relative importance of the Q attribute (Line 1).

Algorithm 1: OveralQFactor Calculation

```
Function: CalculateOveralQFactor (G, Q)
Data: G is refactoring goal. Q is the QualityNode which we want to compute the OveralQFactor of.
Result: Returns OveralQFactor of the Q.
begin
    factor = 0;
    parent = find_parent(G, Q);
    QQ = findQualityStereoType(G, Q);
    if parent is empty then
        factor = 1;
    end
    else
        child_factor =0;
        foreach child in parent.qualities do
            child_factor = child_factor + findQualityStereoType(G, child).importanceFactor;
        end
        relative_importance = (QQ.importanceFactor/child_factor);
        factor = relative_importance * CalculateOveralQFactor(G, parent);
    end
    return factor;
end
```

The difference between *NormalMetric* before and after performing refactoring is multiplied by the calculated *OveralQFactor* for each quality attribute existing in the refactoring goals. The sum over all quality attributes of *OveralQFactor* times the difference between *NormalMetric* before and after refactoring is denoted by W in this paper. If W is greater than zero, the impact of the refactoring pattern on the project is positive. A negative value for W shows the negative effect of the refactoring pattern on the project. In cases where several refactoring patterns are applicable, the refactoring pattern with a higher W has higher priority. Algorithms 2-3 demonstrate the calculation of W in a project by using the refactoring goals.

Algorithm 2 computes the gain or loss corresponding to the increase or decrease in the value of a quality attribute attained by performing a refactoring pattern. At the beginning of the algorithm, the amount of subW (added value) and metricSum are set to zero (Lines 2-2). If the number of metrics used to measure a quality attribute is more than one, we compute and use their average (Lines 2-2). We use *EvalFormula* function for computing the value of the metric formula before and after performing refactoring on the *Model* (Line 2). The *EvalFormula* function is given a

Model, a *MetaModel*, a *Pattern*, and a formula and given the amount of that formula in the model after performing the *Pattern* on the *Model*. If the user does not provide an input *Pattern* for the function, the function evaluates the given formula on the current version of the *Model*. The value of subW equals to the product of the improvement measured by metrics (*metricSum*) by the overall importance of that quality attribute (*OveralQFactor*) (Line 2 of Algorithm 2 does this computation). If the quality attribute has some subQuality attribute, we compute the subW for each of them and add them to the overall subW computed before (Line 2).

Algorithm 3 computes subW for all quality attributes which are affected by performing a refactoring pattern P. This algorithm iterates on the instances of the Goal class and tries to calculate the added value for each of these instances (see Figure 4). In Line 3 of this algorithm, the added (subtracted) value of increasing (decreasing) a quality attribute by performing SA refactoring pattern with regard to the refactoring goals for all affected quality and sub-quality attributes is computed and added to the amount of W.

Algorithm 2: Calculating W for a specific goal

```
Function: CalculateSubW (Model, MetaModel, P, G, goal)
Data: Model we want to perform refactoring on, the MetaModel of the Model, P is the Architecture Refactoring
      Pattern, G is the refactoring goals graph, goal is the quality we want to compute subW for
Result: Returns subW of the goal in graph G.
begin
    subW = 0;
    metricSum = 0;
    foreach metric in goal.metrics do
        metricSum = metricSum + EvalFormula(Model, MetaModel, P, metric.metricFormula) - EvalFormula(Model,
          MetaModel, null, metric.metricFormula);
    end
    metricSum = metricSum/(#goal.metrics);
    subW = metricSum * CalculateOveralQFactor (G, goal.node);
    foreach subgoal in goal.subgoals do
        subW = subW + CalculateSubW (Model, MetaModel, P, G, subgoal);
    end
    return subW;
end
```

Algorithm 3: W Calculation

```
Function: CalculateW (Model, MetaModel, P, G)
Data: Model we want to perform refactoring on, the MetaModel of the Model, P is the Architecture Refactoring
      Pattern which we want to compute W for. G stands for refactoring goals.
Result: Returns W of Pattern P.
begin
    W = 0;
    foreach goal in P.Goal do
        W = W + CalculateSubW(Model, MetaModel, P, G, goal);
    end
    return W;
end
```

3.4.2. Decision Making and Refactoring Component

Refactoring can be seen as a kind of model transformation. In this type of transformation, the source and target models are the same. Some of the M2M languages such as ATL support this type of transformation. In the refactoring case, the *refining* keyword should be used instead of the *from* keyword in the header form of the ATL rules.

```
1 create OUT : S-Model refining IN : T-Model;
```

Unlike the transformation rules for which the input model is read-only and the output model is write-only, in the refining mode the input and output models are identical and rules make inline changes to the input model.

In the previous section, we presented an algorithm for computing W based on refactoring goals and refactoring patterns. As mentioned in that section, the value of W shows the priority of the corresponding refactoring pattern.

The rule priority is not implemented in the majority of M2M languages including the latest version (2010) of ATL. We suggest an approach to overcome this problem. We convert every refactoring rule to a file and check its applicability with the M2M standalone library. These files are sorted based on their calculated W values. The refactoring process starts with the refactoring pattern having the highest W. If the pattern with the highest W is not applicable, the next pattern with highest W is examined. If a refactoring pattern is matched and performed on

the input model, the refactoring process starts again from the pattern with the highest W. This process continues until no other refactoring pattern with $W > 0$ exists.

Algorithm 4, shows the refactoring process which is described in this paragraph. In Line 4-6, we form an array (Wpat) to store the computed W for each pattern. The Wpat array is sorted by the value of the W for each pattern (Line 8). We first perform the patterns with the highest W on the model. The while loop continues until no further changes are applicable to the input model (Line 10). If the added value of performing a pattern is greater than zero the pattern is performed on the model (Line 11). *Find_Pattern* searches for refactoring profile instances by name and returns the one found as the result (Line 12). Generate_ATL method generates the ATL rule from a UML profile instance (Line 12). Run_ATL method in Line 13 runs the generated *ATL* script on a *Model* with regard to the model *MetaModel*. If the *Model* and *NewModel* were not equal to each other (Line 15), we add the *pattern* to the list of refactoring opportunities.

The approach of making an ATL file and executing that file by using ATL Java library and ATL VM is used in some of the other tools such as AML [19].

Algorithm 4: Checking SA Refactoring Patterns on an Architecture Instance

```
Function: CheckRefactoring(Patterns, Goals, Model, MetaModel)
Data: Patterns: a list of refactoring patterns, Goals of refactoring, Model instances, MetaModels in Ecore which
      specifies meta-model of the Model.
Result: List of refactoring opportunities.
begin
    Array Wpat[];
    Array Opportunities[];
    foreach pattern in Patterns do
        array WW = {pattern.name, CalculateW (pattern, Goals)};
        Wpat[] = WW;
    end
    Wpat = Sort_Array(Wpat);
    while true do
        foreach pattern in Wpat do
            if pattern[1] ¿ 0 then
                ATL = Generate_ATL(Find_Pattern(pattern[0]));
                NewModel = Run_ATL(ATL, Model, MetaModel);
                if NewModel ¡¿ Model then
                    Opportunities[] = pattern[0];
                end
            end
        end
    end
    return Opportunities;
end
```

4. Case Study

One of the methods of evaluating software engineering approaches is gauging their practicality in real environments [63]. In this section, a case of SA refactoring (which is described by ACME ADL) is shown.

4.1. Introducing the Case Study

Our goal in this case study is to perform refactoring on a system which is described by ACME ADL [20]. ACME ADL is frequently used to model the architecture of software systems. One of the major goals in designing ACME ADL is to provide an ADL to support interchangeability among ADLs.

User activities in our framework are divided into two levels: *Expert* level and *Regular User* level. The authors of this paper perform the activities at the expert level, and the Online Music Distribution (OMD) developers and stakeholders perform the activities in regular user level, including architecture description and specification of the refactoring goals.

In the expert level of the case study, we model some of the SA refactoring patterns and bad smells. We try to model the refactoring patterns and bad smells which are frequently performed on the ADLs. For this purpose, we implement the rename refactoring and the merge refactoring which are two of the most used refactorings on the software artifacts (according to [44]) in our case study. To show an example of implementing the bad smell using our framework, we described the high coupling bad smell using our framework. The high coupling can decrease the

maintainability of the software artifacts and increase their complexity [45]. By the way, the framework can also describe larger and more complex patterns by using the structures introduced in it.

For evaluating the framework at the regular user level, we perform the activities at that level in a system designed for delivering digital music content to the user, named OMD. We introduce the architecture of OMD, the refactoring goal of OMD according to the need of the stakeholders, and the benefit of performing refactoring on OMD in this case study. For measuring the effectiveness of the refactoring on the OMD project, we survey three different change scenarios on it and evaluate each of them using some measure factors introduced in Section 4.5. We also measure the time complexity of carrying out the refactoring patterns on the OMD project in Section 4.6.

4.2. Expert Level Parts

4.2.1. Defining ACME (Source) Meta-Model

The first step in refactoring architecture based on model transformation is the definition of the source meta-model. The meta-model of the ACME ADL in EMF Ecore Tool [59] is shown in Figure 9. These meta-models come from the definition of the ADL in [20]. The meta-model of the ACME ADL consists of 16 related classes. The relations between classes are in the form of specialization and composition. The *Element* class is the heart of this ADL. Almost every object in this ADL is specialized from this class. This class has a name attribute and has composition relation to *Property* and *Representation* classes. In other words, every *Element* has a name and some *Property* or *Representation*. The *Property* class attaches a name-value tuple to an *Element* in ACME ADL. The properties in this language are used like tags in the UML model. The *Binding* class binds component ports from a source component (compSrc) to a destination component (compDest). Another important class in ACME ADL is the *Attachment* class. This class attaches a port, a connection, or a role to a component. For more investigation on the ACME ADL, one can refer to ACME ADL documents [20].

The meta-model of ACME ADL is saved into a file named *ACME.ecore*.

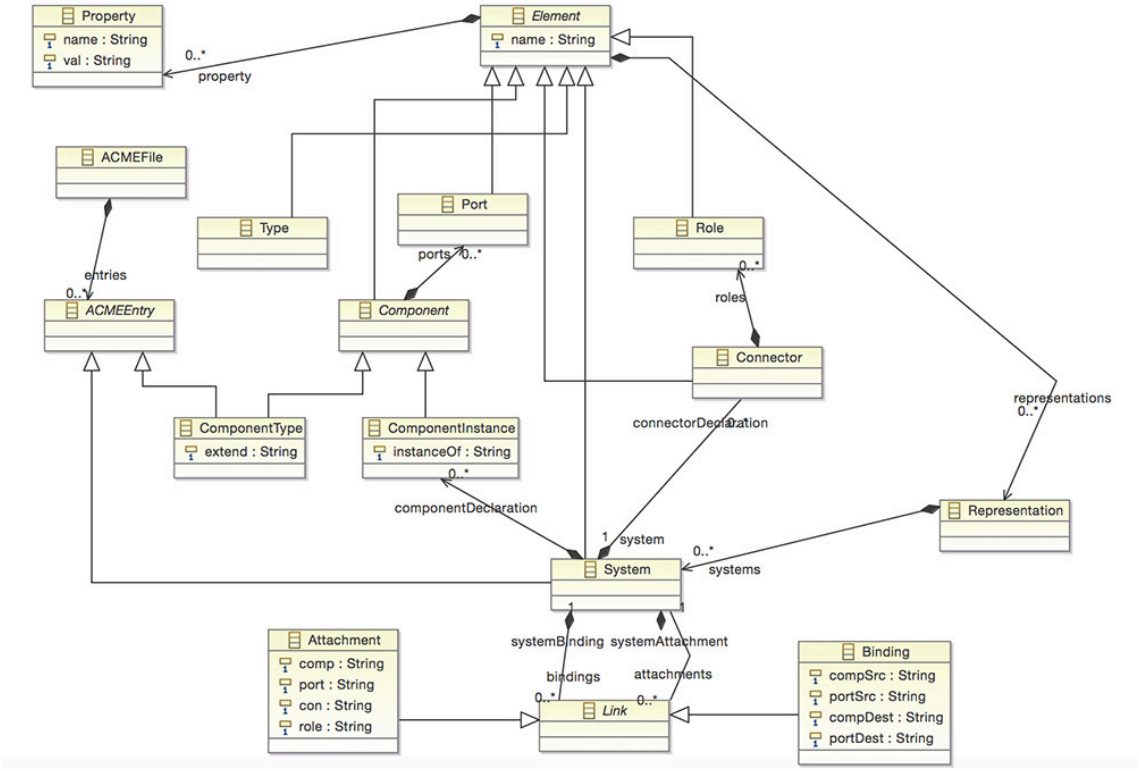

Figure 9: ACME Meta-Model in EMF Ecore Tool [59]

4.2.2. Defining ACME UML Profile

The next step in applying our framework is defining a UML profile. In some domains such as the software product line, a predefined UML profile exists (see e.g. [66]). We failed to find a suitable UML profile for ACME in the literature. We, therefore, created our own UML profile to support ACME ADL. Figure 10 shows the designed

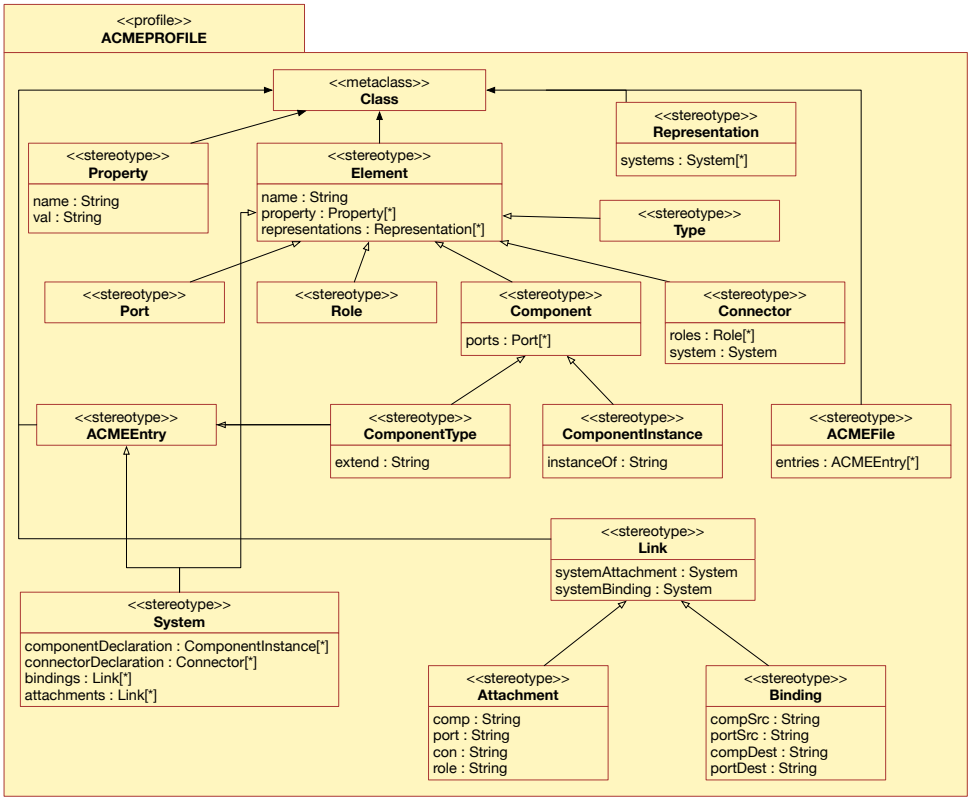

Figure 10: Context UML Profile of the ACME ADL

UML profile. This UML profile has been created based on the ACME meta-model (Figure 9) and supports one to one mapping of the elements between two models without losing any information during the transformation.

We designed this profile by using the four-step guideline explained in Section 3.2.1. The first step in the guideline is the selection of the base meta-class. We select the class meta-class for supporting Element, Property, Representation, Link, ACMEEntry, and ACMEFILE classes in ACME ADL (see Figure 9). That is because, the class meta-model is the best matching and closest syntax meaning class in UML to that classes. Other classes in ACME ADL are derived from these classes. We use the UML inheritance mechanism to support inheritance among classes in the ACME ADL.

The second step in the guideline is determining the attributes of the selected classes (and types thereof). This activity is trivial. We look at ACME ADL and add the required attributes to the designed classes.

In the third step, we check all constraints on the base (class meta-class) and designed class. There is no contradiction among classes in our case study. There are no explicit or implicit contradictions in our designed UML profile.

And finally we check the association between designed classes. The association relations between designed classes are simple and have no contradictions with UML association rules.

The Ecore model of the UML profile is extracted by using EMF [59] and saved into a file named *ACMEProfile.ecore*.

4.2.3. Defining ATL Transformation Rules from Source Model to UML-Profile Model

Due to the one to one mapping of the elements between the ACME ADL and ACMEProfile, the mapping rules are simple.

The ATL transformation rule converts each instance of the ACME ADL into an instance in the UML profile. The ACME ADL meta-model which was defined in Section 4.2.2 and the ACME Profile meta-model which we defined in Section 4.2.1 is used as input to our transformation rules. ATL uses these meta-models for performing the transformation.

A part of the rules for transforming ACME into ACME Profile are as follows. In these rules, we map each element of the ACME ADL to an element in the ACME Profile.

```
1 ...
2 create OUT : ACMEPROFILE from IN : ACME;
```

```
3
4 rule Element2EelementProfile{
5   from m : ACME!Element
6   to a : ACMEPROFILE!Element (
7     name <- m.name
8     property <- prop
9     representations <- reps
10    ),
11    prop : distinct ACMEPROFILE!Property
12    foreach(p in m.property)(
13      name <- p.name,
14      val <- p.val
15     ),
16    reps : distinct ACMEPROFILE!Representations
17    foreach(r in m.representations)(
18      systems <- r.systems
19    )
20 }
21 ...
```

In the code snippet above, each instance of *Element* in ACME is transformed to the Element class in ACMEProfile. Each *Element* in ACMEProfile has three parameters to set: *name*, *property*, and *representations*. The name parameter can be set directly from the input element name (m.name). For assigning *property* and *representations* we iterate on the instances of the *property* and *representations* in selected Element m. The full list of rules to transform an instance of ACME ADL to ACMEProfile appears in A.

The elements used in designing ACME meta-model can support several types of ADLs [20]. In this case study, we define a transformation from ACME to ACMEProfile. This transformation is one to one. However, it is possible to transform other ADLs such as Wright and Rapide to the ACMEProfile. An approach for mapping elements of the Wright and Rapide to the ACME by using the property mechanisms of the ACME is introduced by Garlan *et al.* in [20]. As we mentioned before, by considering the properties of a specific domain and packing them into a UML profile, we can support refactoring on a large number of ADLs in that domain.

4.2.4. Defining ATL Transformation Rules from UML Profile to Source Model

This step is similar to the one described in the previous section. As a result, We do not repeat the rules needed for transforming from ACMEProfile instances to ACME ADL here.

4.2.5. Defining Architecture Refactoring Patterns

One of the most important steps in refactoring SA is defining refactoring patterns. The patterns include *Bad Smells*, *Design Patterns*, and *Architectural Styles*. To define refactoring patterns one should apply the UML profile designed to store the refactoring patterns' information in our framework. Applying the UML profile can be done by using tools such as EMF [59] and Papyrus [14].

In this section, we explain some of the refactoring rules in detail to show the applicability of our framework in defining refactoring rules.

Rename Refactoring

One of the common refactorings performed on the models is rename refactoring. Rename refactoring is usually used by other refactoring patterns. Renaming an element should be done by taking into consideration all relations of the element to the other elements. Referring to the UML profile for ACME ADL, the name of an element is used in the Element, Attachment, and Binding classes. Rename refactoring should modify these classes accordingly for the renaming pattern to be free from side effects and conflicts. Figure 11 shows the rename refactoring rule.

Before introducing the rename refactorings' ATL rule, we present two helpers, which are used in the refactoring rule. These two helpers specify the element we want to rename, together with its new name. Other patterns can use the rename refactoring pattern by changing the values of these helpers.

```
1 helper def : oldName : String = 'oldName';
2 helper def : newName : String = 'newName';
```

The ATL rule of the rename refactoring pattern with regard to ACMEPROFILE meta-model is as follows:

```
1 rule RenameElement(){
2   do {
3     thisModule.RenameElement1();
4     thisModule.RenameElement2();
5     thisModule.RenameElement3();}
6 }
7 lazy rule RenameElement1{
8   from
```

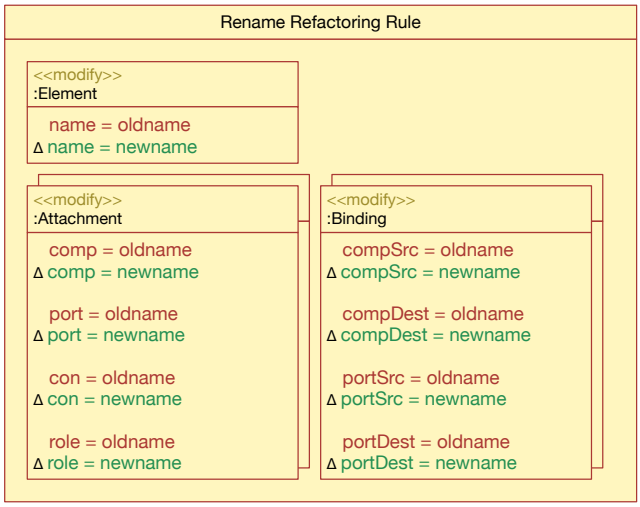

Figure 11: RenameElement Rule

```
9     e : ACMEPROFILE!Element (ACMEPROFILE!Element.AllInstances()->select(h | h.name = thisModule.
          oldName))
10  to
11    t : ACMEPROFILE!Element (t.name <-  thisModule.newName())
12 }
13 lazy rule RenameElement2{
14  from
15    a : ACMEPROFILE!Attachment (ACMEPROFILE!Attachment.AllInstances()->select(h | h.comp =
          thisModule.oldName | h.port = thisModule.oldName | h.con = thisModule.oldName | h.role =
          thisModule.oldName))
16  to
17    t :  ACMEPROFILE!Attachment (
18    if(a.comp = e.name.oldName()) then t.comp <- thisModule.newName() or
19    if(a.port = e.name.oldName()) then t.port <- thisModule.newName() or
20    if(a.con = e.name.oldName()) then t.con <- thisModule.newName()  or
21    if(a.role = e.name.oldName()) then t.role <- thisModule.newName())
22 }
23 lazy rule RenameElement3{
24  from
25    b : ACMEPROFILE!Binding (b.AllInstances()->select(h | h.compSrc = thisModule.oldName | h.
          compDest = thisModule.oldName | h.portSrc = thisModule.oldName | h.portDest = thisModule.
          oldName))
26  to
27    t : ACMEPROFILE!Binding (
28    if(b.compSrc = e.name.oldName()) then t.compSrc <- thisModule.newName() or
29    if(b.compDest = e.name.oldName()) then t.compDest <- thisModule.newName() or
30    if(b.portSrc = e.name.oldName()) then t.portSrc <- thisModule.newName()  or
31    if(b.portDest = e.name.oldName()) then t.portDest <- thisModule.newName())
32 }
```

The type of this rule is call rule. This means that this rule is triggered by calling. *RenameElement* rule calls three other lazy rules to rename an element. In the *from* block of the RenameElement1 rule, we find every element in the ACMEProfile instance which has a *name = oldName* or related to the *oldName* in a way (For example, *Binding* and *Attachment* class using the name attribute in their attributes). We also find the related classes in *Binding* and *Attachment* by using RenameElement2 and RenameElement3 in the code fragment above. After selecting the related article, we change the value of the attribute that contains the *oldName* to the *newName*.

The impact of this refactoring pattern on software quality attributes is determined by the type of the development team and the development approach [3]. Rename refactoring in an incautious team may lead to increased complexity and reduced understandability. In the ideal situation, rename refactoring should increase the readability and understandability of software. Rename refactoring is usually used by other refactoring patterns and is rarely used on its own. In the standalone usages of the rename refactoring pattern, the normal amount of Flesch-Kincaid Grade Level (FCGL) [35] metric can be used. FCGL is a number from -3.40 to +100 [35] and shows the average readability of the old and new name. We use the maximum and minimum amount of the FCGL to calculate the normal value of this metric. The FCGL metric belongs to *readability* sub-quality attribute of the ISO 9126 standard.

Component Merge Refactoring

One of the methods of decreasing design complexity is combining different structures. The merge pattern combines two components and creates a new component with a new name. The connection between the merged components has to be dropped in order to perform this refactoring pattern. The refactoring rule should create a new component and bind the roles, the ports and the connections of the merged components to it. We use rename refactoring to migrate the roles, ports and connections from the merged components to the new component. Figure 12 shows the main activity in merging two components in ACME profile.

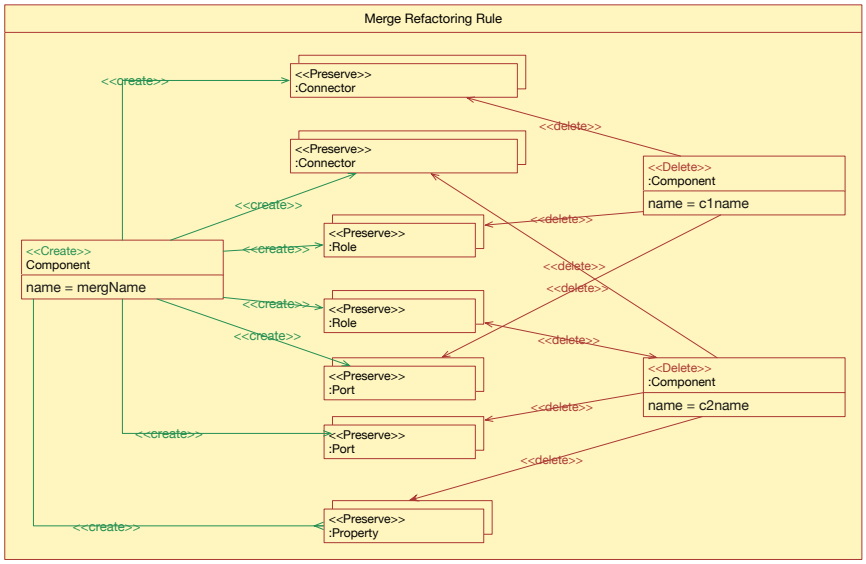

Figure 12: MergeComponent Rule

We utilize two helpers in the component merging process. These helpers aid in declaring the components which we want to merge and provide a mechanism for using the merge pattern in the other patterns. The two helpers, getC1Name and getC2Name, are as follows:

```
helper def : getC1Name : String = 'c1name';
helper def : getC2Name : String = 'c2name';
```

The ATL Rule of this pattern is as follows:

```
1  lazy rule MergeComponent{
2    from
3      c1 : ACMEPROFILE!Component (ACMEPROFILE!Component.AllInstances()->select(e | e.name =
           thisModule.getC1Name),
4      c2 : ACMEPROFILE!Component (ACMEPROFILE!Component.AllInstances()->select(e | e.name =
           thisModule.getC2Name),
5      con : ACMEPROFILE!Connector,
6      attachment : ACMEPROFILE!Attachment
7      (ACMEPROFILE!Attachment.AllInstances -> select(e| e.con = conn.name and e.comp = c1.name) ->
           select(e | e.con = conn.name and e.comp = c2.name)))
8    to
9      drop,
10     c3 : ACMEPROFILE!Component (
11     c3.name <- c1.name + '_' + c2.name
12     c3.Port <- c1.Port->union(c2.Port)
13     c3.property <- c1.property->union(c2.property)
14     c3.representations <- c1.representations->union(c2.representations)
15     )
16   do {
17     thisModule.oldName <- c1.name ;
18     thisModule.newName <- c3.name;
19     thisModule.RenameElement();
20     thisModule.oldName <- c2.name;
21     thisModule.RenameElement();}
22 }
```

The type of this rule is lazy. In the code snippet above, we first select two components which we want to merge (Line 3-4) and then select the connection and attachment between them (Line 5-7). In order to perform the merge, we create a new component with a new name (Line 9-14). The *property* and *representations* of the new component

are the union of the *property* and *representations* of the two components we want to merge (Line 11-13). Finally, we drop the two merged components and the relations between them (Line 15).

After merging two components, we call rename refactoring to rename every instance of the old component to the new name (Line 17-21). For this purpose, we set the helpers and call the rename refactorer. Lines 19 and 21 call RenameElement rule.

Merge refactoring is not used on its own. The impact of the merge pattern on the quality attributes is determined by the reason behind performing the merge operation. This refactoring is usually used to decrease the complexity, and improve the performance of systems. Sometimes, merge refactoring is used to merge components with similar functionality in a system. Because of that, in the description of this refactoring pattern, we do not specify any quality attribute goals. In the next section, we introduce a pattern which uses this pattern to decrease the complexity of software.

Removing High Coupling

High coupling is one of the well-known bad smells in designing software architecture. A large number of relations between two components decreases performance, and increases design complexity drastically. This bad smell can be avoided by defining a threshold on the number of the relations which can exist between two components. We survey a simple case of the bad smells in design. In this case, if the amount of Coupling Between Object (CBO) [9] exceeds a threshold, the *HighCouplingBadSmell* refactoring pattern is matched and the transformation rule can be executed. This pattern uses the merge pattern, which we described in the previous section. The coupling threshold defined in the helper named Threshold is as follows:

```
1 helper def : Threshold : Integer = 5;
```

The ATL Rules of the pattern are as follows:

```
1 rule HighCouplingBadSmell{
2   from
3     c1 : ACMEPROFILE!Component,
4     c2 : ACMEPROFILE!Component,
5     attachment : ACMEPROFILE!Attachment
6     (
7       ACMEPROFILE!Attachment.AllInstances ->select(a |
8       ACMEPROFILE!Connector.AllInstances ->select(con |
9       ACMEPROFILE!Attachment.AllInstances ->select(a2 |
10      c1.name != c2.name and
11      (a.comp = c1.name and a.con = con.name and a2.comp = c2.name and a2.con = con.name
12      or
13      a.comp = c2.name and a.con = con.name and a2.comp = c1.name and a2.con = con.name
14      )
15      ))))
16  to
17    if(attachment.Count > 2*Threshold) then (thisModule.getC1Name <- c1.name, thisModule.
          getC2Name <- c2.name)
18  do {
19    thisModule.MergeComponent();}
20 }
```

To find the component with high coupling, we need to do some join in the meta-model of the ACMEProfile. In the code snippet above, we perform the join by multiple uses of the *select* function of the OCL. In line 7, we select all instances of the attachment. In line 8, we add all instances of the connector. In line 9, we add all instances of the attachment. Now we can select all instances of connector between two attachments which are located between two components C1 and C2 (Line 11 and 13).

The code in line 3 to 15 is run for each pair of the components in the architecture. For each pair we see in line 17, we check the refactoring condition. If the condition is true, the helper for doing merge refactoring is set.

We use the CBO metric [9] to measure the complexity of software. The metric formula is A/N. A is the number of connection between components and N is the number of components. We write this formula according to the model meta-model as follows:

$$\frac{Attachment.AllInstances().Count}{Component.AllInstances().Count * 2}$$

We simply divide the number of attachments in the *Model* by the number of the components that exists in it. Because each connection between two components has two rows in the *Attachment* class, we divide the result by two. We set *metricMin* of this metric to 10 and *metricMax* to zero. A Higher value of the CBO metric indicates a higher rate of coupling between two components (Lower CBO value is better). This metric is a member of the maintainability quality attributes.

We use the *postCondition* variable to check the behavior preservation of the *remove-high-coupling* rule. As we described before, *postCondition* can access the model before and after refactoring using *Old* and *New* input model. These input models are provided by our framework as inputs of the *postCondition* variable. Remove high coupling refactoring is behavior preserving if the refactoring rule merges two high coupled components and their relations to the other components. In the code snippet below we check these conditions.

```
1 New!Component.AllInstances->select(newcomp |
2 Old!Component.AllInstances->select(c | c.name = newcomp.name)->isEmpty() and
3
4 Old!Component.AllInstances->select(oldcomp |
5 New!Component.AllInstances->select(c | c.name = oldcomp.name)->isEmpty() and
6
7 Old!Attachment.AllInstances->select(a |
8 a.comp = oldcomp.name and
9 New!Attachment.AllInstances->select(b | b.con = a.con and b.comp = newcomp.name)->notEmpty()
10 ) and
11 newcomp.Port->includesAll(oldcomp.Port) and
12 newcomp.property->includesAll(oldcomp.property) and
13 newcomp.representations->includesAll(oldcomp.representations)))->notEmpty()
```

In the above code fragment, we first select the newly added component which did not exist in the Old model (Lines 1-2). In Lines 4-5, we select the old removed components from the Old model. We then check the condition of behavior preservation in Lines 7-13. The refactoring is behavior preserving if every connection in the Old model is present in the New model (with new component names, Lines 7-10) and every *Port*, every *Property*, and every *Representation* also exists in the New model (with new component names, Lines 11-13).

Figure 13 shows the application of refactoring pattern profile in Papyrus [14] for defining *HighCouplingBadSmell* pattern.

Figure 13: Applying Refactoring Profile in Papyrus [14]

4.3. Regular User Level

In this section we describe an instance of the ACME ADL and the way we can perform refactoring on it using our framework.

4.3.1. Introducing OMD Architecture

Figure 14 shows the architecture of the *OMD* which is described using ACME ADL notation. The purpose of this project is to distribute online music to registered users. The project has a mobile app as well. In the three-tier architecture of this system, data, logic, and presentation are separated. Based on the commonality of the functionality of the UI controller, a component to support UI requests and events is considered in the architecture of the system. The web-based application and the mobile app web service use this component. The relation between the UI and BI controllers is higher than the normal and acceptable relation between two components. The tight coupling between these two components creates some problems. For one, the maintainability of the BI controller becomes harder. For another, a small modification of the BI controller component makes a huge impact on the UI controller. Besides, adding new UI actions to the UI controller is not easy.

4.3.2. Refactoring Goals in OMD

As mentioned before, refactoring goals specify the amount of stakeholders' satisfaction of improving each quality attribute. In each project, based on the type of the project and its domain, a particular set of quality attributes have become more important. An approach to prioritizing refactoring goals is suggested throughout this paper. Each quality and sub-quality attribute of software are specified by a factor from 0 to 10. These factors are later used in our decision-making algorithms to compute W for each refactoring pattern.

Maintainability has a large priority in the OMD system. The system is immature; therefore, many change requests are received over time. The changes are mainly in the form of adding new functionality or fixing a bug. Software usability and reliability are also important in this project. Finally, the project security should be at an

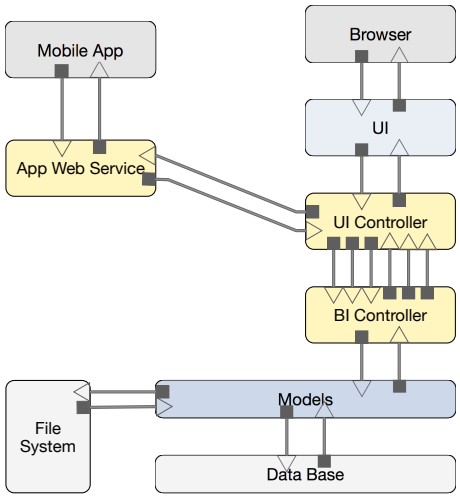

Figure 14: The Architecture of a OMD Application.

appropriate level and the accuracy of the response to the user request should be at a high level. We illustrate the refactoring goals using a graph in Figure 15. The values in the graph should be expressed using the Refactoring Goals Context Profile. For this, we should apply the refactoring goal profile with the five quality attributes which are represented in Figure 15 as *qualities* variable. This figure specifies the importanceFactor of each quality attribute of the ACME architecture instance. The applied UML profile is stored in a file and is given as an input to the framework later.

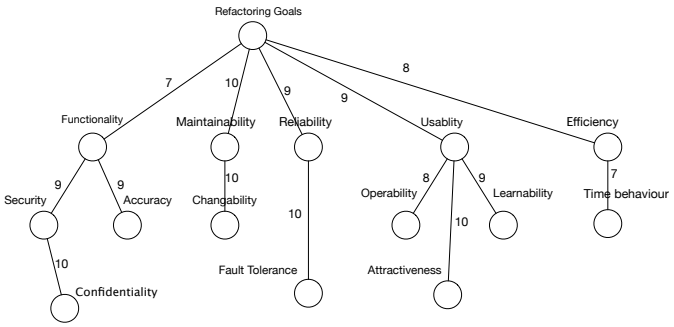

Figure 15: Refactoring Goals of OMD

4.4. Tool Level

4.4.1. Choosing Refactoring Patterns and Performing Refactoring on Architecture

The final step in applying our framework is to perform refactoring on the architecture instances. For performing refactoring on the SA architecture, we need to develop a tool to perform the designed algorithms on each instance of the source ADL. Figure 16 shows the primary version of the designed tool to perform SA refactoring based on model transformation[1].

The first input of our tool is the source meta-model. The Ecore meta-model of ACME is available online and we can use it in our case study. The second input is the designed UML profile meta-model. One can use tools like EMF [59] to produce UML profiles in Ecore format. In our example, we used EMF [59] and exported the UML profile meta-model to Ecore format. The transformation rule from source ADL to UML profile model and UML profile model to the source ADL, are the next input of our tool. We present the complete list of the rules needed to do the transformation from ACME to ACMEProfile in A. The ACMEProfile to ACME transformation rules are similar to the one we represent in A. The next input in our tool is architecture refactoring patterns. The file *ACMEpatt.uml* consists of several instances of the *Refactoring Pattern* profile we introduced before.

[1]A copy of ARiA implementation available at https://github.com/tanhaei/ARiA

The first five inputs in our tool are at the Expert level. These inputs are the same for refactoring each instance of the ACME ADL. In the final version of our tool, the regular users need to select their ADL language among the provided ADLs. In this regard, the user faces one of the following scenarios:

- Enough material exists for supporting refactoring on the selected ADL.
- Some of the inputs required for performing refactoring (from the first five) are not available.

In the former case, the user can do the SA refactoring without engaging in the technical aspects of defining refactoring patterns and other activities at the expert level. In the latter case, the user can do the task of experts and provide the needed material using the approaches introduced in our framework or simply withdraw from performing the refactoring.

The sixth input to our tool are the architecture instances we want to perform refactoring on. In our example, the architecture of the OMD application in ACME is the sixth entry. The last input of the tool is refactoring goals. The user should apply the *Refactoring Goal* profile to provide this input to our tool. The last two entries in our tool are in regular user level. These entries may differ in every refactoring activity on a specific ADL instance. In the final version of our tool, we will include an interactive tool to get the refactoring goals from users. Our interactive tool translates the tree which is formed by the users, to some UML profile instances.

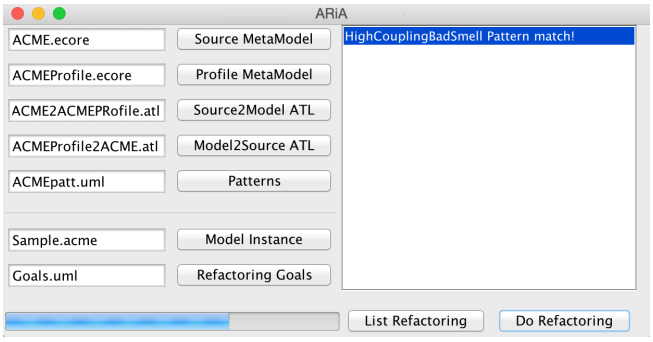

Figure 16: The ACME Architecture Instance in ARiA Framework.

When we complete the input needed for performing refactoring, the tool can list the refactoring opportunities. Based on the definition of the HighCouplingBadSmell pattern and the specified threshold, the HighCoupling-BadSmell pattern matches the ACME instance introduced in Figure 14. The computed W for the *HighCoupling-BadSmell* pattern in this case study is:

$W = \frac{10}{10+9+9+8+7} * ((6 * \frac{28/(8*2)-10}{0-10} - 3) - (6 * \frac{40/(9*2)-10}{0-10} - 3)) = 0.065 \text{ ¿ } 0$

which is a positive number. Figure 16 shows the refactoring opportunities which our refactoring tool finds in the OMD architecture.

To perform the pattern on the architecture instance one should press the Do Refactoring button. By ordering the refactoring, the refactoring pattern is applied by merging the *UI Controller* and *BI Controller* components, which have a coupling ratio greater than the defined threshold. After performing refactoring activities including merging and renaming some classes on the OMD architecture, our tool checks the post condition of the refactoring to ensure the behavior preservation of the OMD architecture. Executing post condition of the *remove high coupling* pattern on the *Old* and *New* architecture of the OMD returns *true*, meaning the pattern did not change the behavior of the OMD architecture. As a result, our framework commits the changes performed by refactoring. Figure 17 shows the OMD architecture after performing refactoring on its architecture.

To perform merge refactoring on the OMD architecture, one needs to know about the traceability of the architecture to artifacts in other levels of abstraction. This is because by performing modifications on the SA, all related artifacts at other levels of abstraction are affected. Figure 18 shows a mapping of the OMD architecture to lower levels of abstraction such as design and code. By merging the *UI Controller* and *BI Controller* components, *UI Controller Class* and *BI Controller Class* should be merged to keep the consistency between the architecture level and the design level. In order to support modification done on these two classes, every class related to these classes should also be investigated and may be modified to support changes done on these two classes. For example, classes inherited from one of these two classes should now inherit from the merged class. By changing the classes in the design level, code written to support these classes should also be modified. As one can see in Figure 18, even small changes on the SA can impose large changes on the other levels of abstraction.

In Figure 18, some of the components of the architecture are external components and cannot be mapped to lower levels of abstraction such as design and code. These types of components cannot simply be refactored in the SA. For example, one cannot refactor *Database* component or merge it to other components of the OMD project.

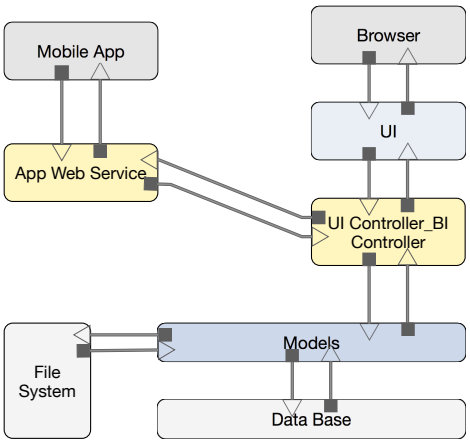

Figure 17: Architecture of the OMD after Refactoring

Table 2: Required effort to perform requirements in the original (ORG) and refactored (REF) version of the OMD

Requirement	TRP		NOC		NOA		NOF		LOC	
	ORG	REF	ORG	REF	ORG	REF	ORG	REF	ORG	REF
Requirement 1	530h	440h	13	7	4	2	44	27	13k	7k
Requirement 2	213h	215h	5	5	6	4	68	53	4.5k	4k
Requirement 3	426h	441h	6	5	6	4	136	141	11.5k	12k
Average	389h	365h	8	5.6	6	3.3	82.6	73.6	9.6k	7.6k

4.5. Software Architecture Refactoring Effects on the OMD Project

In this section we want to evaluate the effect of architecture refactoring on the maintainability of the OMD system. In order to measure maintainability improvement in the OMD project, we proposed three change scenarios in the OMD system. We want to measure the effect of the SA refactoring on the speed and accuracy of performing modifications on OMD. The developers of the OMD were asked to perform these changes on the OMD system:

- Requirement 1: Providing offline playback of the music for premium users.
- Requirement 2: Users be able to share their playlist with their friends.
- Requirement 3: Users be able to make their own radio station.

We measure these factors for evaluating the speed and accuracy of modification performed by the OMD developers team to implement the needed requirement:

- Time needed to perform requirement (**TRP**): For measuring this time we used the log of the project management system.
- Number of the changed classes (**NOC**): We measured NOC by analyzing project SVN data.
- Number of newly added classes (**NOA**): This value can be measured like NOC in OMD project.
- Number of faults found after release (**NOF**): We measured NOF by analyzing project bug-tracker data. Bug-tracker data is logged by the OMD test group.
- Number of changed and added lines of code (**LOC**): We measured LOC by analyzing project SVN data.

In Table 2 we compare the required effort to perform needed requirements on the OMD before and after performing refactoring on it. As one can see in Table 2, the majority of the factors we measured on the OMD project are improved in the refactored version of the OMD.

As a result of decreasing coupling between components of the OMD, the number of classes, which should be changed (NOC) and the number of new classes, which should be added to support new requirements (NOA) are decreased. Decreases in the number of the changed and added classes also decrease the number of faults found after release.

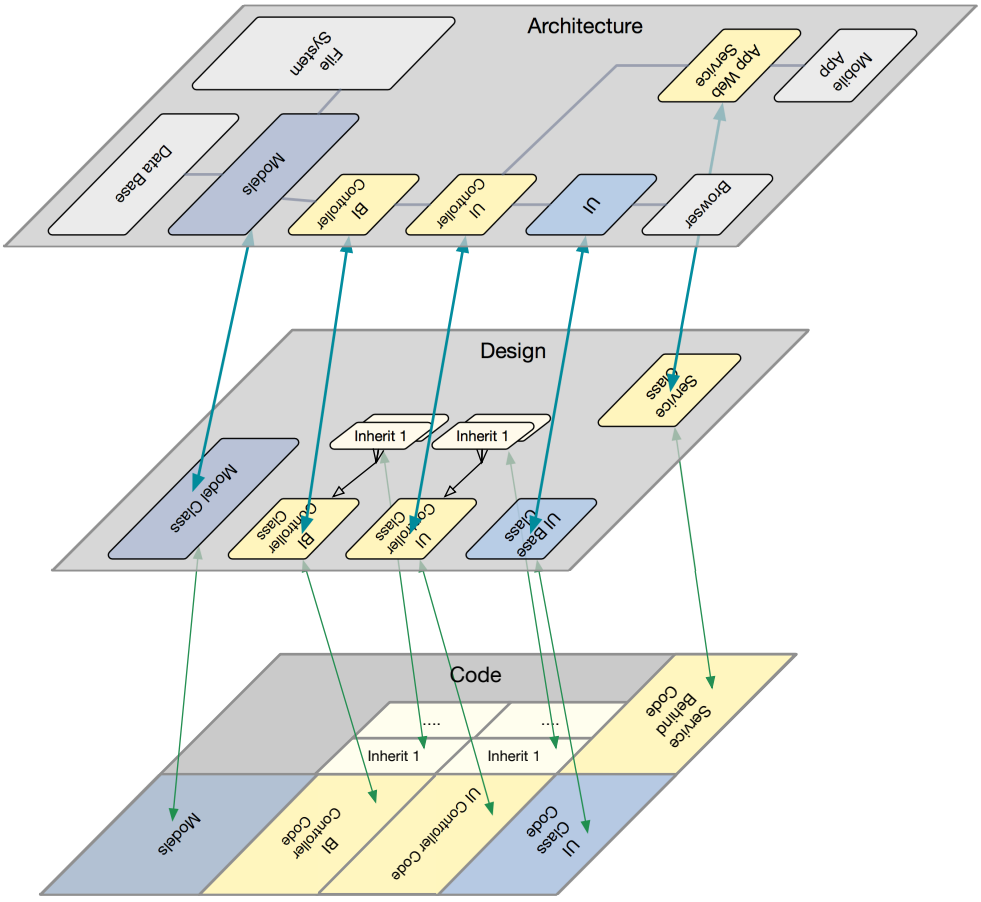

Figure 18: Mapping of the OMD Architecture to the design and code levels

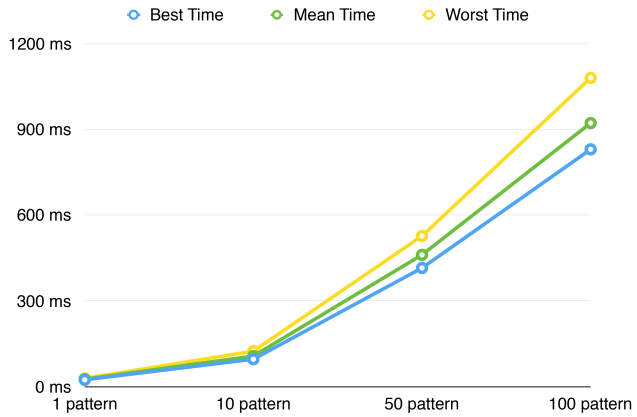

Figure 19: The Result of Experimenting on the Number of Refactoring Patterns in the case of the OMD Architecture

The amount of improvements in performing requirement 1 are much stronger than the other two requirements. This is because implementing requirement 1 strongly relyed on the two *UI controller* and *BI controller* components in the OMD system. Implementing other two requirements needs performing modifications on the *Models* and *App Web Service* components. Because we do not change this component, we do not expect much improvement in performing these requirements on the OMD system.

However, our experimental evaluation of the OMD system shows that the time, the number of changed classes, and the number of faults in the refactored version of the OMD are generally decreased. The amount of improvements in the change scenarios that relate to the refactored location are much greater than others.

4.6. OMD Refactoring Performance

In this section, we want to evaluate the performance of our proposed approach. We want to measure the performance of the approach in performing refactoring on the OMD which is described using ACME ADL. We use the artifacts developed in Section 4.2 and 4.3 to conduct our experiments. We used a Macbook Pro Retina Mid 2013 with 8 GB of RAM utilizing a CPU clocked at 2.3 GHz to run our experimental tests. We want to measure the performance of the proposed approach in these different conditions:

- When the number of refactoring patterns are varied.

- When the number of the pattern goals and the number of metrics in each pattern goal are varied.

Performance of the framework when the number of patterns are varied

We want to examine the performance of our approach in finding refactoring opportunities when 1, 10, 50, and 100 refactoring patterns are defined in the framework. We replicate the *Remove High Coupling Refactoring* pattern proposed in Section 4.2.3 zero, 9, 49, and 99 times and execute the framework using these replicated patterns. *Remove High Coupling Refactoring* pattern only has one goal, which contains one metric. We executed the proposed framework 10 times in each condition (one pattern, 10 patterns, and so on) and show the best, the worst, and the mean execution time (in milliseconds). In Figure 19 the X-axis specifies the number of patterns used to find refactoring opportunities and the Y-axis shows times that the proposed framework required to find the first refactoring opportunities on the input models. The number of refactoring patterns on the SA rarely exceeds 20. As one can see in Figure 19, the performance of our proposed framework in case of 10 and 50 patterns is acceptable. We used a pattern containing only one goal and one metric in this experiment. In the next experiment, we varied the number of goals and metrics and calculate time complexity in each condition.

Performance of the framework when number of metrics and goals are varied

In this section, we want to examine the performance of the proposed framework when we have a different number of goals and metrics in each refactoring pattern. In this section, we use the *Remove High Coupling Refactoring* pattern proposed in Section 4.2.3 with some modifications. For testing one goal and one metric condition, we use the *Remove High Coupling Refactoring* pattern as is (It has one goal and one metric in its definition). For testing one goal and five metrics, we replicate the metric used in the *Remove High Coupling Refactoring* four times. For testing five goals and five metrics, we replicate the *Remove High Coupling Refactoring* goal four times. For testing five goals and 10 metrics, we first replicate the metric in the goal one time, and then replicate the goal four times. We finally replicate the proposed pattern (in each condition) 10 times and run the refactoring framework on it. We executed the proposed framework 10 times in each condition and recorded the execution time. Figure 20 shows the

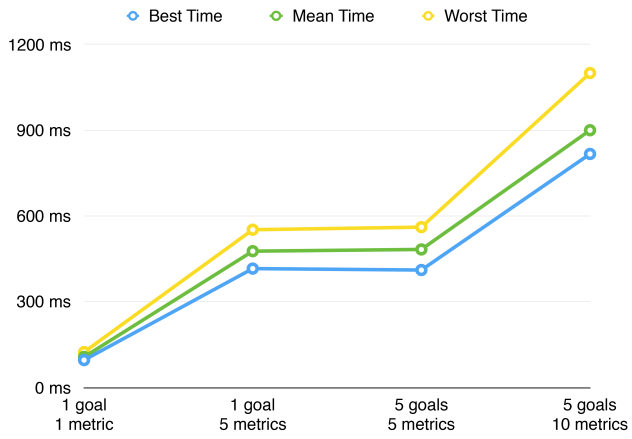

Figure 20: The Result of Experimenting on the Number of Goals/Metrics in the case of the OMD Architecture

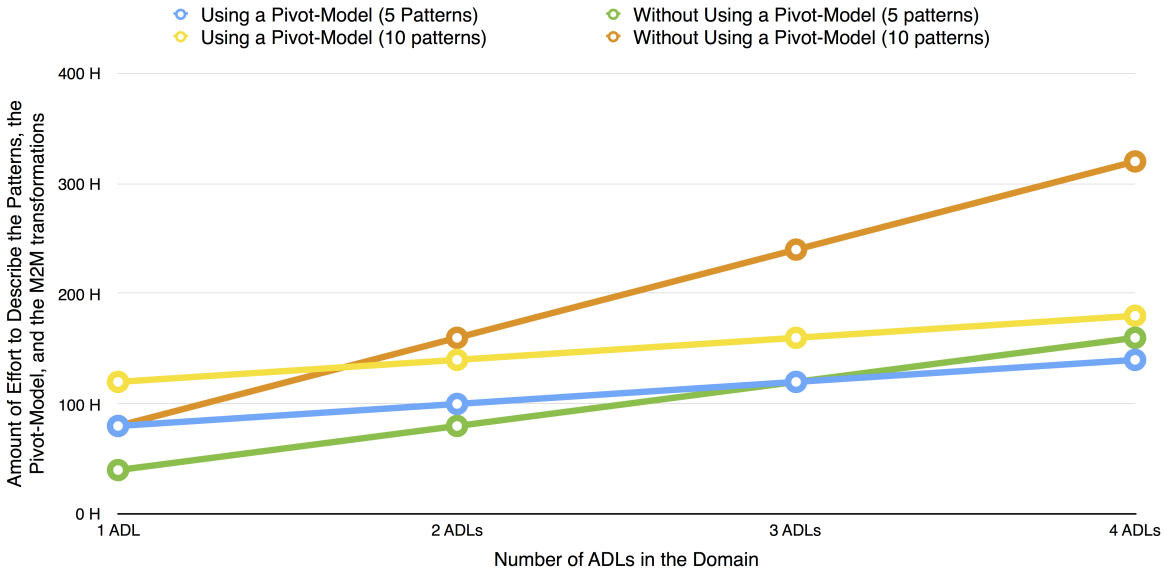

Figure 21: The effect of Using a Pivot-Model in the Amount of effort need to Describe the Refactoring Patterns

result of our experiment. As one can see in this figure the running time of the algorithm to find the first refactoring opportunities depends on the total number of the metrics proposed in the patterns defined in it. This is because, running time for the pattern with one goal and five metrics is approximately the same as the pattern with five goals and five metrics. The complexity of the proposed framework is linear with regard to the number of patterns' metrics defined in it.

4.7. The Usability of the Framework

At first glance, the suggested refactoring pattern may seem to be too complex to be really usable by architects. But when one takes a closer look at this framework, one notices that a high amount of reuse is achieved using it. Table 3 shows the number of times each part of the framework is created to perform SA refactoring on n architecture instances. The majority of the tasks in this framework are done only once. For example, the source meta-model is defined just once. After that, the meta-model of the source model can be reused many times. Having reusable parts in the refactoring process creates an opportunity to form a refactoring repository for each source model.

4.7.1. The Usability From the Expert User Viewpoint

The expert users perform the majority of the works in our framework. They define the pivot-model, the M2M scripts for transforming from a source model to the pivot-model and vice versa, and the refactoring patterns. In our framework, we used a pivot-model to provide refactoring pattern reuse in a particular domain. We performed some measurement on the amount of time needed for defining five and ten refactoring patterns in a domain containing

Table 3: The number of refactoring artifacts builds used in performing the refactoring process on n architecture instances.

Artifact Name	#
The Source Meta-Model	1
The UML Profile and its corresponding Generated Meta-Model	1
ATL Transformation Rule from Source to UML Profile	1
Refactoring Patterns, Bad Smells and Architecture Styles	1
Architecture Instances	n
The Refactoring Goals	n
ATL Transformation Rule from UML Profile to Source	1

different number of ADLs. Our experiment shows that, by increasing the number of refactoring patterns and ADLs in a domain, using a pivot-model becomes more economical. For example, Figure 21 shows that using a pivot-model for describing five refactoring patterns in a domain becomes economical if the number of ADLs in that domain is larger than two while describing ten refactoring patterns in that domain become economical if the number of ADLs in that domain be greater than one. Our experiment also shows that using a pivot-model for describing a domain with only one ADL is not economical.

4.7.2. The Usability From the Architect's Viewpoint

From the architect's viewpoint, our framework does not differs from other refactoring tools. The architect should provide the architecture of the system she/he wants to refactor, and the goal of the refactoring by applying the profile designed in our framework. The process of transforming the ADL to a pivot-model, checking the conditions of each refactoring pattern, performing the refactoring activities defined in the refactoring pattern, checking the post-conditions of the pattern, and converting the pivot-model to the original ADL are all done by the framework in the background.

4.8. Precision and Recall of the Proposed Framework

The precision of a refactoring framework means the fraction of the correct refactoring opportunities the framework finds, and the recall of a refactoring framework means the fraction of the refactoring opportunities the framework finds. In case of low precision, the refactoring framework may suggest wrong refactoring opportunities and in case of low recall, the refactoring framework may fail to find refactoring opportunities. In this research, we proposed a UML profile for defining the refactoring patterns. This UML profile enables expert users to define refactoring patterns using the ATL and OCL languages. ATL and OCL have no limitation on describing refactoring patterns in detail. The precision and recall of the proposed framework directly depend on the details level of the refactoring patterns defined in it. As a result, the precision and recall of the proposed framework directly depend on the modeler of the refactoring patterns.

The amount of the precision and recall for every refactoring pattern can be calculated using the following formulas:

$$\text{Precision} = \frac{\#\{\{\text{Relevant refactoring}\} \cap \{\text{Founded refactoring}\}\}}{\#\{\text{Founded refactoring}\}}$$

$$\text{Recall} = \frac{\#\{\{\text{Relevant refactoring}\} \cap \{\text{Founded refactoring}\}\}}{\#\{\text{Relevant refactoring}\}}$$

One can compute precision and recall for every refactoring pattern described in the proposed framework. For an example, in the case of OMD refactoring, the precision of the *Remove High Coupling Refactoring* pattern is 100 percent because all refactoring opportunities found by this pattern are relevant. The amount of recall in this case is also 100 percent because all merge refactoring opportunities were found by the pattern. For another example, consider *Folding Rule* in Section 3.2.2. The precision of this rule is 100 percent, because all found refactoring opportunities are true. The amount of recall in the case of *Folding Rule* is not 100 percent. This is because the

pattern is not matched for methods without parameters. *Folding Rule* recall equals the amount of duplicated methods that have parameters divided by all duplicated methods in a program.

The amount of precision and recall can reveal the implementation quality and effectiveness of the patterns. However, the amount of precision and recall may vary from a project to another. While a low amount of precision and recall for a pattern is a strong indicator of faulty description of that pattern, a high amount of precision and recall cannot guarantee absolute correctness of that pattern in other refactoring situations.

5. Threats to Validity

Internal validity: Our findings in Section 4.5 show a relation between refactoring on the OMD and reduction in the number of faults, release time, and changed/added LOC. It is possible that during performing refactoring on the OMD other components and classes of the OMD change in such a way that facilitate performing required changes on the OMD.

Construct validity: Construct validity issues arise when one makes errors in measurement. One of the threats to validity in this regard is the correctness of the formulation of the refactoring patterns. The mistakes in the description of the refactoring patterns can lead to an incorrect conclusion about whether or not to perform refactoring on a part of the software architecture.

We collect the information about fault and development time from OMD local project management tool. It is possible that a developer failed to log the end-time of performing a requirement correctly. It is also open to consideration that some of the fault of the OMD were not logged in the OMD project management tool.

One may make errors in performing replication in Section 4.6. As a result, it is possible that a wrong number of replications cause misinterpretation in this section.

External validity: We performed the refactoring activities on the OMD project (An instance of the ACME ADL). Due to the commercial nature of this project, its artifacts cannot be accessed publicly. However, we supplied major artifacts related to the OMD project in the case study. These artifacts can help in repeating the experiments and validating it.

The result of Section 4.6 is obtained by performing replication on a pattern in our framework. The architecture of the OMD is available in Figure 14. We provide the replicated pattern and the goals/metrics used in it in the case study. We also provide an implementation of the proposed framework on the internet. As a result, we believe that the experiments in Section 4.6 are replicable.

6. Discussion

Estimating in quantitative terms the benefit of refactoring is probably one of the biggest challenges in software engineering. In this research, we proposed an algorithm to assess the impact of refactorings on the software quality. However, this algorithm suffers from the lack of appropriate metrics. We need to develop new metrics to measure the effects of SA changes on other levels of abstraction. Our refactoring framework can only use quantitative metrics which may limit its usage in some situations.

Architecture refactoring only makes sense if it can be later mapped to source code and artifacts in other levels of abstraction. The problem is that when we perform refactoring in architecture, we can easily violate many semantic and type rules of programming languages. For example, we cannot move a method to a class that already has a method with the same signature. As a result, these preconditions (traceability and consistency of the architecture to code) should be checked before performing refactoring on the SA. The team that performs refactoring on the architecture should use appropriate tools such as the one developed by Adersberger *et al.* [2] to ensure that the refactoring does not raise problems such as inconsistency in the lower levels of abstraction. Adersberger *et al.* introduced a UML-Based approach to check traceability and consistency of the architecture written in a specific ADL to code [2]. They provide a UML profile to map each element of the architecture to code.

We defined our case study on the structure (Components and Connectors) of a simple software system (OMD). However, our framework is not limited only to this software architecture viewpoint (Structural). One can use our refactoring framework to refactor artifacts representing other viewpoints on the SA such as deployments, physical, and behavioral viewpoints.

In this paper, we provide several examples of defining refactoring pattern conditions using OCL. A Large number of patterns can be specified using the OCL language. Guennec *et al.* [24] suggest a guideline for specifying patterns using OCL. They define several design patterns such as Observer, Composite, and Visitor in their work. In our framework, expert users can use OCL alongside ATL to define refactoring patterns. However, it is possible that some refactoring patterns cannot be specified using OCL.

Our framework helps in performing software architecture refactoring in two ways:

Table 4: Comparison of the architecture and model refactoring frameworks

Metric	Our framework	Reimann [52]	Rose [53]	Ruhroth [54]	Ivkovic [28]	Moha [43]
Supported ADLs	All ADLs	All DSL, Models	All Models	All BNF-Style Languages	UML	All Models
User can Define refactoring patterns?	Yes (Expert Users)	Yes (DSL Designer)	Yes	Yes	Yes	Not Specified
User can define Refactoring Goals?	Yes	No	No	No	Yes	No
Implementation Tool	ATL, OCL	EOL, Role Model	Generic Programming	Not Specified	UML, OCL	Kermeta, Model Typing
Check Refactoring Correctness?	Yes	Yes but manually (User should add constraints to the model)	No	Yes using after templates	Yes but manually (using post-conditions)	No
Support Multiple Architecture View?	No	No	No	No	No	No
Graphical Representation?	No	Yes (Role Models)	No	No	No	No
Metrics Type	Quantitative	Not Uses Metrics	Not Uses Metrics	Not Uses Metrics	Qualitative	Not Uses Metrics

- **Discovering refactoring opportunities on the software architecture:** Enlargement of the software architecture or the sophisticated nature of some of the software architecture patterns makes it difficult to check refactoring conditions manually. In this regard, our framework facilitates finding refactoring opportunities on the software architecture.
- **Performing refactoring activities on the software architecture:** Our framework also helps in performing refactoring activities on the SA. We introduced merge refactoring on the OMD project in the case study section; however, performing refactoring even on a seemingly simple refactoring pattern such as merge refactoring is far from straightforward. One needs to modify the related classes to the merged components including Ports, Attachments and Connections to do merge refactoring.

7. Related Work

Domain-Specific Languages provide the ability to define semantics and schemas in a specific domain. These languages use a graphical or textual representation to avoid modeling complexity and increase ease of use. Graphical representation decreases syntax errors and increases trust level over the models, and facilitates finding errors in the models [56].

Different UML profiles for describing software architecture have been created. The definition of AADL [16] contains a UML profile, which can describe real-time and embedded systems. Goulao *et al.* [22] suggest a profile for ACME ADL. Amirat *et al.*, describe the hardness of using the UML profile for architecture and define a UML profile for describing software architecture in [48]. Oquendo *et al.*, develop a UML profile to describe π-ADL [48]. π-ADL is a Formal Architecture Description Language. Several techniques for describing architecture using UML are compared and the weaknesses and strengths of the techniques are surveyed in [4]. Giachetti *et al.*, suggested a method for automatically producing UML profiles for DSMLs [21]. Some instances of using a UML profile to describe a specific domain can be found in the domain of software product line [66], real-time applications [5], and software architecture description [31]. These instances show the strengths of using UML profiles in modeling different domains.

The idea of using model transformation to investigate ADLs has appeared in the literature. Kim used model transformation to describe and investigate patterns on the software design [34]. He used a role-based meta-modeling language to define design patterns.

Reimann *et al.* [52] suggest a framework for reusing model refactoring across multiple domain-specific languages. They used the role model to model refactoring rules and implement their framework in the EMF. Similar to our framework, they used three levels of abstraction to model and perform refactoring on the DSL models. We compare their framework to our framework in Table 4

Rose *et al.* [53] also used generic programming to define model-driven activities such as refactorings, animations, transformations, etc. They used EOL (a language similar to OCL) to define manipulations on models. The idea of

their work is to bring an approach into model management to raise the level of abstraction of the specification of model manipulation. In general, we can divide the work in their framework into three levels similar to our framework: tool level, designer level, and user level. However, because they aimed to manage every kind of manipulation on the model, they failed to check special requirements of model refactoring such as checking behavior preservation and refactoring goals.

A generic refactoring language for specification and execution named ReL is introduced by Ruhroth *et al.* [54]. The generic refactoring language can specify and run refactoring patterns on a given code. Before-template, precondition, refactoring actions, and after-template can be specified using the ReL language. Moha *et al.* [43] introduced a generic model refactoring approach. They used Kermeta, an extension of the MOF meta-model standard, and Model Typing to specify refactoring patterns and to apply them on a domain of similar models.

Malavolta *et al.* [39] provide a tool named DUALLY to provide interchangeability among ADLs themselves as well as UML. They used ATL to transform the models to each other. Similarly to our approach, they used a middle model (UML profile) called A_0 to maintain the source model information. The mapping between elements in source and target model in their tools is specified by a graphical tool. The mapping rule in ATL is generated automatically when the user specifies the mapping between two models. By the similarities and relations which actually exist between our framework to DUALLY, the aims of the two frameworks are completely different. Our aim is to provide an infrastructure to support software architecture refactoring while DUALLY is providing interoperability among ADLs. For our purpose (SA refactoring), we provide a mechanism to specify SA refactoring patterns and SA refactoring goals in our framework.

Stevens [50] used OCL to describe refactoring on the models. However, because OCL is free of side-effects and cannot alter the models, he wrote his own Python based tool named SMW [49] to solve this problem. The SMW [49] notation is very similar to the notation used in ATL transformation rules. In ATL transformation language, we can use OCL constraints and ATL declaratives simultaneously. Guennec *et al.* [24] suggest an approach which can be used to specify detailed description of the patterns. They use a modified version of UML and OCL to define pattern constraints.

An applicable technique for describing design patterns using UML diagrams is proposed by Robert France[2] *et al.* [18]. They describe some of the design patterns (e.g. Observer-Visitor style) using the proposed technique. Zdun *et al.* [65] use UML profiles to define architectural patterns. Commonalities of the software architecture are defined by pattern primitives in their work. Pattern primitives are specified using UML Stereotypes. After that, one defines a mapping between pattern primitives and architectural building blocks. *EMF Diff Merge/Patterns* [10] is a proper tool that provides the definition, the alteration, and the evolution of the patterns in the EMF modeling environment. Similar to our work, it uses OCL to define the pattern constraints. This tool can be simply integrated into the EMF modeling tools. However, this tool in the current shape is not suitable for defining refactoring patterns. That is because in establishing refactoring patterns we need to define conditions, the shape of the model after refactoring and the actives needed to perform refactoring on the model. The main idea of using the UML profile to store refactoring patterns comes from [28]. Ivkovic and Kontogiannis [28] introduce a UML profile to capture software architecture refactoring patterns, and a UML profile called Conceptual Architecture View for maintaining architectural information. However, they do not suggest how the refactoring should be implemented or how the source architecture be transformed to their Conceptual Architecture View profile. Using a single UML profile to store information presented in all types of architecture limits the applicability of their framework. A UML profile to define pattern elements is proposed by Dong *et al.* [12]. They used UML Stereotypes and tagged values to specify their UML profile. The relation of the pattern elements to each other is not considered in their pattern specification.

Model refactoring can be seen as design evolution. Each refactoring pattern improves a part of the design. PROGRES [51] is an environment for specifying graph rewriting rules and performing matched rules on the models. Arendt *et al.* developed a tool for in-place EMF model transformations called Henshin [6]. Their work supports endogenous transformation on the EMF models. As a case study, they explain several EMF model refactorings in their work. They define refactoring patterns (and their conditions) in the meta-model of the model they want to refactor. Our way of defining refactoring patterns on the ADLs is similar to the work of Arendt *et al.* [6]. We first model the meta-model of the ADL by a middle model and then define the refactoring condition on the meta-model of this middle model.

A catalogue of refactorings for model-to-model transformations using ATL is defined by Wimmer *et al.* [62]. They used ATL to represent a catalogue of refactorings on the models. This catalogue can facilitate the activity of defining new refactoring rules in our framework.

[2]The late Founder and Editor-In-Chief of SoSyM

8. Conclusion and Future Work

Performing refactoring on software artifacts is a good way to increase their quality attributes [17]. Performing refactoring on code increases the maintainability and understandability of the code. The refactoring process is, however, not limited to source code. Higher levels of software development such as software architecture can also benefit from this process.

Performing refactoring on the SA is a good way to increase software quality before implementation. Identification of software pitfalls before implementation decreases the cost of development drastically. Avoiding the known bad smells in designing architecture and applying appropriate SA styles and patterns decreases the likelihood of project failure and increases the satisfaction of the stakeholders. Based on the above mentioned benefits, performing refactoring before going into implementation phase is quite advisable.

Performing refactoring after implementation may help in finding design bugs and can suggest styles and patterns useful for increasing the quality of the software. Applying a refactoring pattern on an implemented system is a hard task to do. This kind of process has to deal with the traceability of the architecture elements to the source code and can lead to large modifications to the source code, huge changes in documentation and so on.

Applying refactoring at the SA level is not as simple as the code level. The first problem one faces when performing refactoring on the SA is the lack of proper tools to support automatic or semi-automatic refactoring. The diversity of the domains and viewpoints for describing the SA increase the number of ADLs and the ways in which the SA can be described. One cannot realistically hope to design refactoring tools for all different ADLs. To deal with the diversity of ADLs, in this paper we set out to convert every ADL to a middle model and define the refactoring patterns and constraints of each pattern on that model.

While it is possible that more than one ADL describe one domain, the properties of each domain are independent of the ADLs. Each of the ADLs with the same domain can be transformed to a UML profile (middle model) which describes the properties of that domain. Using the patterns defined in that UML profile, the refactoring process can then be performed on the SA.

The impact of SA refactoring on software qualities is wider than the impact of code refactoring. Each refactoring pattern affects some of software quality and sub-quality attributes. In this paper, we introduce a way to define the impact of each refactoring pattern on software quality and sub-quality attributes (**S3**). A metric is used to measure the impact of refactoring patterns on each quality or sub-quality attribute. The fact that we do not have enough metrics to measure software quality attributes is troublesome. One important future work is to define new metrics to quantify different quality attributes.

Refactoring goals describe the aim of the refactoring process. Applying a refactoring pattern which increases system security and decreases system performance where the refactoring goals prioritize performance increase, is not a good choice. Refactoring goals specify the importance of each quality and sub-quality attribute of the software. The refactoring goals should be taken into account when performing SA refactoring.

Each refactoring pattern has an added value to the project. We use W to specify this added value. This W is calculated by considering refactoring goals and the effect of the refactoring patterns on software quality attributes. The metrics measure the impacts of a refactoring pattern on a quality attribute. To homogenize the impact of each metric in calculating the value of W, we introduce the concept of metric normalization. The normal values are between -3 and 3. We assume that the values of the metrics grow linearly. This is not a good assumption for every metric, because some metrics exhibit not-linear behavior.

The aim of this framework is to provide a tool for performing automatic refactoring on the SA. Despite all the benefits that this framework provides, some critical challenges remain unsolved. Difficulties in describing patterns, bad smells, and refactoring conditions using a particular language is the first challenge. The semantics of some patterns is written in natural languages, and it is hard to describe them using known languages (such as OCL). It is obvious that we should find a way to describe these patterns formally, if we want them in automatic analysis tools. The second challenge is the heterogeneity of the metrics. Our suggestion to overcome the problem of metric heterogeneities (normalization of the metrics) is not applicable in the case of non-linear metrics. In this situation a mapping table (between the real and normalized values) can be used. One of the other challenges is the small number of the metrics which can be used to measure the quality attributes with. The fourth challenge comes from transformation languages such as QVT [23] and ATL [29]. These languages do not carry out priority rules. Implementing the priority rules in the 3G languages (such as Java) decreases the flexibility and scalability and increase complexity of developing tools.

The number of important refactoring patterns in each domain is not very large. Focusing on these important refactoring patterns and defining them formally (using OCL) can lead to large improvements. Creating a repository to store the meta-models of ADLs, meta-models of UML profiles for each domain, the description of the patterns in each domain, and M2M transformation rules, is one of the future works of this research. Investigating the architectural styles and finding their relation to software quality attributes is another possible future work.

A. ATL Transformation Rules from ACME ADL to ACMEProfile

```
module ACME2ACMEPROFILE;
create OUT : ACMEPROFILE from IN : ACME;
rule Element2EelementProfile{
  from  m : ACME!Element
  to  a : ACMEPROFILE!Element (
    name <- m.name
    property <- prop
    representations <- reps
    ),
    prop : distinct ACMEPROFILE!Property
    foreach(p in m.property)(
      name <- p.name,
      val <- p.val
     ),
    reps : distinct ACMEPROFILE!representations
    foreach(r in m.representation)(
      systems <- r.systems
    )}
rule ACMEFile2ACMEFilePROFILE {
  from  m : ACME!ACMEFile
  to  a : ACMEPROFILE!ACMEFile (entries <- entry),
    entry : distinct ACMEPROFILE!ACMEEntry
    foreach(e in m.entries)(
    --nothings
    )}
rule PORT2PORT extends Element2EelementProfile{
  from  m : ACME!Port
  to  a : ACMEPROFILE!Port (
    --nothing! inheritance do the job!)
}
rule Type2Type extends Element2EelementProfile{
  from  m : ACME!Type
  to  a : ACMEPROFILE!Type ()
}
rule ROLE2ROLE extends Element2EelementProfile{
  from  m : ACME!Role
  to  a : ACMEPROFILE!Role ()
}
rule COM2COM extends Element2EelementProfile{
  from  m : ACME!Component
  to  a : ACMEPROFILE!Component (
    ports <- thisModule.resolveTemp(ACMEPROFILE!Port.allInstances() -> select(p | p.name in m.
        ports->collect(name))))
}
rule Conn2Conn extends Element2EelementProfile{
  from  m : ACME!Connector
  to  a : ACMEPROFILE!Connector (
    roles <- thisModule.resolveTemp(ACMEPROFILE!Role.allInstances() -> select(r | r.name in m.
        roles->collect(name))),
    system <- thisModule.resolveTemp(ACMEPROFILE!System.allInstances() -> select(r | r.name = m.
        system)))
}
rule CT2PCT extends COM2COM, ACMEEntry{
  from  m : ACME!ComponentType
  to  a : ACMEPROFILE!ComponentType (extend <- m.extend)
}
rule CI2PCI extends COM2COM{
  from  m : ACME!ComponentInstance
  to  a : ACMEPROFILE!ComponentInstance (instanceOf <- m.instanceOf)}
rule REPS2REPS extends Element2EelementProfile{
  from  m : ACME!Representation
  to  a : ACMEPROFILE!Representations (
    systems <- thisModule.resolveTemp(ACMEPROFILE!System.allInstances() -> select(s | s.name in m
        .system->collect(name))))}
rule SYS2SYS extends Element2EelementProfile{
  from  m : ACME!System
  to  a : ACMEPROFILE!System (
    componentDeclaration <- thisModule.resolveTemp(ACMEPROFILE!Component.allInstances() -> select
        (r | r.name in m.componentDeclaration->collect(name)))
    connectorDeclaration <- thisModule.resolveTemp(ACMEPROFILE!Connector.allInstances() -> select
        (r | r.name in m.connectorDeclaration->collect(name)))
```

```
66      bindings <- bind
67      bindings <- attach
68      ),
69      bind : distinct ACMEPROFILE!Binding
70      foreach(b in m.bindings)(
71        compSrc <- b.compSrc,
72        portSrc <- b.portSrc,
73        compDest <- b.compDest,
74        portDest <- b.portDest,
75        systemBinding <- self
76      ),
77      attach : distinct ACMEPROFILE!Attachment
78      foreach(b in m.attachments)(
79        comp <- b.comp,
80        port <- b.port,
81        con <- b.con,
82        role <- b.role,
83        systemAttachment <- self
84      )}
```

References

[1] A. Abouzahra, J. Bézivin, M. D. Del Fabro, F. Jouault, A practical approach to bridging domain specific languages with UML profiles, in: Proceedings of the Best Practices for Model Driven Software Development at OOPSLA, Vol. 5, 2005.

[2] J. Adersberger, M. Philippsen, Reflexml: Uml-based architecture-to-code traceability and consistency checking, in: Software Architecture, Springer, 2011, pp. 344-359.

[3] S. Alshehri, L. Benedicenti, Rankingtherefactoring techniques based on the internal quality attributes, International Journal of Software Engineering & Applications 5 (1) (2014).

[4] A. Amirat, M. Oussalah, et al., Towards an UML profile for the description of software architecture, in: Proceeding of International Conference on Applied Informatics (ICAI'09), 2009, pp. 226-232.

[5] L. Apvrille, J. Courtiat, C. Lohr, P. de Saqui-Sannes, Turtle: a real-time UML profile supported by a formal validation toolkit, IEEE Transactions on Software Engineering, 30 (7) (2004) 473-487.

[6] T. Arendt, E. Biermann, S. Jurack, C. Krause, G. Taentzer, Henshin: Advanced concepts and tools for in-place emf model transformations, in: D. Petriu, N. Rouquette, Ø. Haugen (Eds.), Model Driven Engineering Languages and Systems, Vol. 6394 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2010, pp. 121-135.

[7] B. W. Boehm, J. R. Brown, M. Lipow, Quantitative evaluation of software quality, in: Proceedings of the 2nd international conference on Software engineering, IEEE Computer Society Press, 1976, pp. 592-605.

[8] S. Bosems, A performance analysis of model transformations and tools, Master's thesis, University of Twente (2011).

[9] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering, 20 (6) (1994) 476-493.

[10] O. Constant, Emf diff merge/patterns.

[11] J. Dietrich, C. Elgar, A formal description of design patterns using owl, in: Software Engineering Conference, 2005. Proceedings. 2005 Australian, IEEE, 2005, pp. 243-250.

[12] J. Dong, Y. Sheng, K. Zhang, Visualizing design patterns in their applications and compositions, IEEE Transactions on Software Engineering, 33 (7) (2007) 433-453.

[13] R. G. Dromey, A model for software product quality, IEEE Transactions on Software Engineering 21 (2) (1995) 146-162.

[14] H. Dubois, F. Lakhal, S. Gérard, The papyrus tool as an eclipse uml2-modeling environment for requirements, in: Proceedings of the 2009 Second International Workshop on Managing Requirements Knowledge, MARK '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 85-88.

[15] A. H. Eden, Precise specification of design patterns and tool support in their application, Ph.D. thesis, Publisher not identified (2000).

[16] P. H. Feiler, B. Lewis, S. Vestal, The sae avionics architecture description language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering, in: RTAS 2003 Workshop on Model-Driven Embedded Systems, 2003.

[17] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley Professional, 1999.

[18] R. B. France, D.-K. Kim, S. Ghosh, E. Song, A UML-based pattern specification technique, IEEE Transactions on Software Engineering, 30 (3) (2004) 193-206.

[19] K. Garcés, F. Jouault, P. Cointe, J. Bézivin, A domain specific language for expressing model matching, in: 5ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM09), 2009, pp. 33-48.

[20] D. Garlan, R. Monroe, D. Wile, Acme: an architecture description interchange language, in: Centre for Advanced Studies on Collaborative research, IBM Corp., 1997, pp. 7-22.

[21] G. Giachetti, B. Marín, O. Pastor, Using UML as a domain-specific modeling language: A proposal for automatic generation of UML profiles, in: Advanced Information Systems Engineering, Springer, 2009, pp. 110-124.

[22] M. Goulão, F. B. e Abreu, Bridging the gap between Acme and UML 2.0 for CBD, in: Proceedings of Specification and Verification of Component-Based Systems (SAVSCB'03), workshop at ESEC/FSE 2003, 2003, pp. 75-79.

[23] O. M. Group, et al., Query/view/transformation specification version 1.0, formal/2008-04-03, April (2008).

[24] A. Le Guennec, G. Sunyé, J.-M. Jézéquel, Precise modeling of design patterns, in: UML 2000—The Unified Modeling Language, Springer, 2000, pp. 482-496.

[25] C. Hofmeister, R. Nord, D. Soni, Applied software architecture, Addison-Wesley Professional, 2000.

[26] S. Hussain, Investigating architecture description languages (adls) a systematic literature review, Master's thesis, Linköpings universitet (2013).

[27] ISO/IEC, ISO Standard 9126: Software engineering - product quality, parts 1, 2 and 3 (2001 (part 1), 2003 (parts 2 and 3)).

[28] I. Ivkovic, K. Kontogiannis, A framework for software architecture refactoring using model transformations and semantic annotations, in: Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on, IEEE, 2006, pp. 10-23.

[29] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, Atl: A model transformation tool, Science of computer programming 72 (1) (2008) 31-39.

[30] S. H. Kan, Metrics and models in software quality engineering, Addison-Wesley Longman Publishing Co., Inc., 2002.

[31] M. M. Kandé, A. Strohmeier, Towards a UML profile for software architecture descriptions, in: UML 2000—The Unified Modeling Language, Springer, 2000, pp. 513-527.

[32] S. Kent, Model driven engineering, in: Integrated formal methods, Springer, 2002, pp. 286-298.

[33] R. Khare, M. Guntersdorfer, P. Oreizy, N. Medvidovic, R. N. Taylor, xadl: enabling architecture-centric tool integration with xml, in: System Sciences, 2001. Proceedings of the 34th Annual Hawaii International Conference on, IEEE, 2001, pp. 9-pp.

[34] D.-K. Kim, Design pattern based model transformation with tool support, Software: Practice and Experience 45 (2013) 473-499.

[35] J. P. Kincaid, R. P. Fishburne Jr, R. L. Rogers, B. S. Chissom, Derivation of new readability formulas (automated readability index, fog count and flesch reading ease formula) for navy enlisted personnel, Tech. rep., DTIC Document (1975).

[36] A. G. Kleppe, J. B. Warmer, W. Bast, MDA explained: the model driven architecture: practice and promise, Addison-Wesley Professional, 2003.

[37] K. Lano, J. Bicarregui, S. Goldsack, Formalising design patterns, in: RBCS-FACS Northern Formal Methods Workshop, 1996.

[38] A. Lauder, S. Kent, Precise visual specification of design patterns, in: ECOOP'98-Object-Oriented Programming, Springer, 1998, pp. 114-134.

[39] I. Malavolta, H. Muccini, P. Pelliccione, D. A. Tamburri, Providing architectural languages and tools interoperability through model transformation technologies, IEEE Transactions on Software Engineering 36 (1) (2010) 119-140.

[40] J. A. McCall, P. K. Richards, G. F. Walters, Factors in software quality. volume-iii. preliminary handbook on software quality for an acquisiton manager, Tech. rep., DTIC Document (1977).

[41] T. Mens, P. Van Gorp, A taxonomy of model transformation, Electronic Notes in Theoretical Computer Science 152 (2006) 125-142.

[42] T. Mikkonen, Formalizing design patterns, in: Proceedings of the 20th international conference on Software engineering, IEEE Computer Society, 1998, pp. 115-124.

[43] N. Moha, V. Mahé, O. Barais, J.-M. Jézéquel, Generic model refactorings, in: Model driven engineering languages and systems, Springer, 2009, pp. 628-643.

[44] E. Murphy-Hill, C. Parnin, A. P. Black, How we refactor, and how we know it, Software Engineering, IEEE Transactions on 38 (1) (2012) 5-18.

[45] J. Offutt, A. Abdurazik, S. R. Schach, Quantitatively measuring object-oriented couplings, Software Quality Journal 16 (4) (2008) 489-512.

[46] E. Oliva, Interactive graphical maps for infocenter via model to model transformation, Eclipse-IT 2009 (2009) 104.

[47] W. F. Opdyke, Refactoring object-oriented frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign (1992).

[48] F. Oquendo, Formally modelling software architectures with the UML 2.0 profile for π-adl, ACM SIGSOFT Software Engineering Notes 31 (1) (2006) 1-13.

[49] I. Porres, A toolkit for model manipulation, Software and Systems Modeling 2 (4) (2003) 262-277.

[50] I. Porres, Model refactorings as rule-based update transformations, in: P. Stevens, J. Whittle, G. Booch (Eds.), UML 2003 - The Unified Modeling Language. Modeling Languages and Applications, Vol. 2863 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2003, pp. 159-174

[51] A. Radermacher, Support for design patterns through graph transformation tools, in: Applications of Graph Transformations with Industrial Relevance, Springer, 2000, pp. 111-126.

[52] J. Reimann, M. Seifert, U. Aßmann, On the reuse and recommendation of model refactoring specifications, Software & Systems Modeling 12 (3) (2013) 579-596.

[53] L. Rose, E. Guerra, J. De Lara, A. Etien, D. Kolovos, R. Paige, Genericity for model management operations, Software & Systems Modeling 12 (1) (2013) 201-219.

[54] T. Ruhroth, H. Wehrheim, S. Ziegert, Rel: A generic refactoring language for specification and execution, in: Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on, 2011, pp. 83-90. doi:10.1109/SEAA.2011.22.

[55] T. L. Saaty, What is the analytic hierarchy process?, Springer, 1988.

[56] B. Selic, A systematic approach to domain-specific language design using UML, in: Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC'07. 10th IEEE International Symposium on, IEEE, 2007, pp. 2-9.

[57] S. Siraj, L. Mikhailov, J. A. Keane, Priest: an interactive decision support tool to estimate priorities from pairwise comparison judgments, International Transactions in Operational Research 22 (2013) 217-235.

[58] O. A. Specification, UML 2.0 infrastructure specification.

[59] D. Steinberg, F. Budinsky, E. Merks, M. Paternostro, EMF: eclipse modeling framework, Pearson Education, 2008..

[60] S. Vestal, Metah programmer\'s manual (1996).

[61] M. Völter, T. Stahl, J. Bettin, A. Haase, S. Helsen, Model-driven software development: technology, engineering, management, John Wiley & Sons, 2013.

[62] M. Wimmer, S. M. Perez, F. Jouault, J. Cabot, A catalogue of refactorings for model-to-model transformations., Journal of Object Technology 11 (2) (2012) 1-40.

[63] R. Yin, Case Study Research: Design and Methods, Applied Social Research Methods, SAGE Publications, 2009.

[64] Y. Yu, J. Mylopoulos, E. Yu, J. C. Leite, L. Liu, E. D'Hollander, Software refactoring guided by multiple soft-goals, in: 1st workshop on Refactoring: Achievements, Challenges, and Effects, in conjunction with the 10th WCRE conference 2003, IEEE Computer Society, 2003, pp. 7-11.

[65] U. Zdun, P. Avgeriou, Modeling architectural patterns using architectural primitives, in: ACM SIGPLAN Notices, Vol. 40, ACM, 2005, pp. 133-146.

[66] T. Ziadi, L. Hélouët, J.-M. Jézéquel, Towards a UML profile for software product lines, in: Software Product-Family Engineering, Springer, 2004, pp. 129-139.

[67] O. Zimmermann, Architectural refactoring: A task-centric view on software evolution, IEEE Software (2) (2015) 26-29.

Please cite this article using:

Mohammad Tanhaei, A model transformation approach to perform refactoring on software architecture using refactoring patterns based on stakeholder requirements, AUT J. Math. Comput., 1(2) (2020) 179-216
DOI: 10.22060/ajmc.2020.17541.1027