Fighting evil is not enough when refactoring metamodels: promoting the good also matters

Oussama Ben Sghaier
University of Montreal
oussama.ben.sghaier@umontreal.ca

Houari Sahraoui
University of Montreal
sahraouh@iro.umontreal.ca

Eugene Syriani
University of Montreal
syriani@iro.umontreal.ca

ABSTRACT

In model-driven engineering, metamodels are central artifacts that allow to capture domain concepts and build domain-specific languages. However, bad design decisions, continuous changes, and the evolution of requirements may introduce bad smells and deteriorate the quality of metamodels. Refactoring metamodels is a complex task as it should be performed according to many conflicting quality attributes while maximizing the removal of smells. In this paper, we propose a generic automated approach based on a multi-objective heuristic search to refactor metamodels. The process aims at generating a set of refactoring recommendations with various quality trade-offs from which the modeler can choose the most appropriate for her context. We evaluate the efficiency of our approach with a user-based experiment, on time to perform understandability and extendibility tasks, as well as the correctness of the task output. Our results show that, globally, considering trade-offs between quality and smell removal is significantly better than focusing on smell removal alone. The observed difference is statistically significant for the extendibility but only partially for the understandability.

KEYWORDS

model-driven engineering, software quality, refactoring, multi-objective optimization, search-based software engineering.

ACM Reference Format:
Oussama Ben Sghaier, Houari Sahraoui, and Eugene Syriani. 2022. Fighting evil is not enough when refactoring metamodels: promoting the good also matters. In *The 37th ACM/SIGAPP Symposium on Applied Computing (SAC '22), April 25–29, 2022, Virtual Event,* . ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3477314.3507053

1 INTRODUCTION

Metamodels are central to abstraction and automation in model-driven engineering (MDE). They represent the structural essence of models by defining domain concepts, their relations, and their features. They are used in most MDE activities, such as language engineering, model transformation, code generation, consistency, and conformance validation [9].

Given this high importance of metamodels, they should be designed carefully by taking into account several quality factors such as reusability, extendibility, and understandability [6, 21]. In practice, metamodels require several and continuous changes to satisfy new requirements and other maintenance activities. However, these modifications may hinder their quality by making them unnecessarily more complex, less understandable, or less flexible. Because many other MDE artifacts depend on metamodels, this eventually negatively affects productivity, and increases fault-proneness and maintenance costs of the development [1, 20, 32].

Maintenance has always been a challenging task in software engineering as it is expensive and effort consuming [16]. Metamodels should be regularly refactored to improve their design and maintain their consistency. Therefore, full or partial automation of refactoring alleviates the burden on modelers. Metamodel refactoring consists of restructuring the design by introducing some changes to its structure without altering its intrinsic semantics [13]. One approach to refactoring consists of detecting smells and correcting them using refactoring operations [24, 26].

Many approaches were proposed for metamodel and model refactoring. They are based on formal methods [15], model transformations [33], or a learning process from preexisting examples [14, 25]. These proposed methods use different techniques to detect refactoring opportunities without worrying about the quality factor, which is the main goal of refactoring. The refactoring operation should not be performed haphazardly but, instead, be based on well-defined objectives, such as improving some quality criteria.

Bettini et al. [8] presented a quality-driven framework for detecting and resolving metamodel smells. They define a static mapping between design smells and quality attributes. Then, refactoring a metamodel consists of removing all the bad smells that have an impact on the target quality attributes. Nevertheless, we believe that removing all the design smells is not necessarily the best solution, as it may not necessarily lead to the best values of quality attributes. In fact, quality attributes are potentially conflicting [2, 17]. Improving one quality criterion could lead to the degradation of another. For instance, introducing too many elements to a metamodel to improve its flexibility and extendibility may harm its understandability and increase the cost of maintenance. Thus, a refactoring solution should find the best trade-off with respect to the target quality attributes.

In this paper, we propose a generic and automated approach based on multi-objective heuristic search to refactor metamodels. Our approach aims at finding a compromise between the conflicting objectives to improve the quality of the metamodel. It recommends a set of non-dominated refactoring solutions with various quality trade-offs. Then, the modeler can pick a solution according to her context and preferences. We evaluate the efficiency of our approach

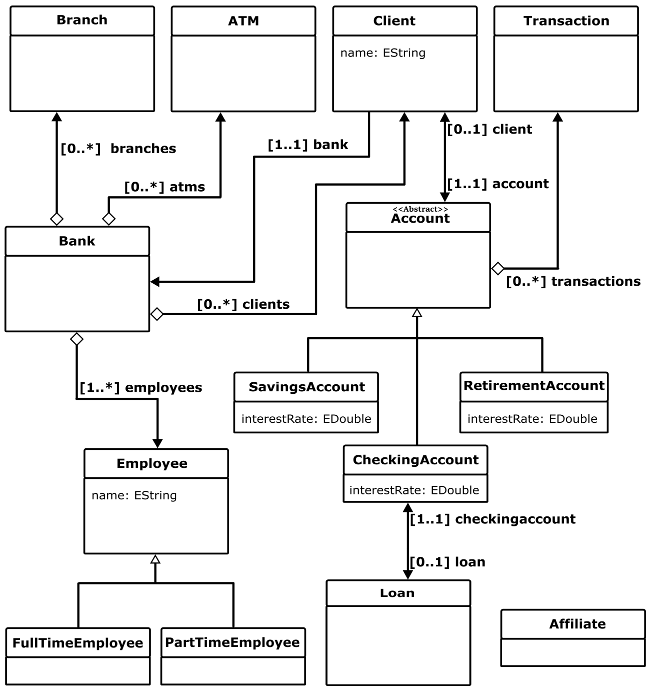

Figure 1: Metamodel for bank management systems

using a user-based experiment. We measure the time to perform tasks related to understandability and extendibility as well as the correctness of the task output. The results show that our approach outperforms other alternatives and generates relevant refactoring solutions that improve the quality of the metamodel. The observed difference is globally statistically significant considering trade-offs between quality and smell removal. More specifically, the statistical significance was complete for extendibility but partial for understandability.

The remainder of this paper is organized as follows. We introduce a motivating example in Section 2. We then detail the proposed approach for metamodel refactoring. First, we explain the evaluation framework of metamodels in Section 3. Then, we explain how we perform the refactoring as an optimization problem in Section 4. In Section 5, we report the evaluation results and we discuss different threats related to our experiments in Section 6. Finally, we outline the related work in Section 7 and conclude in Section 8.

2 MOTIVATING EXAMPLE

We present the design of a simplified metamodel for bank management systems as a motivating example. As shown in Figure 1, `Bank` is the central class that is composed of automated teller machines (`ATMs`) and `Branches`. It employs `FullTime` or `PartTime` `Employees`. A bank has `Clients` who hold different types of `Accounts`: `Checking`, `Savings` or `Retirement`. An account logs a list of `transactions`. A checking account can have a `Loan` attached to it. Finally, `Affiliate` represents the employees' affiliation. This metamodel contains many types of design smells [29], which are the result of poor design decisions. We briefly describe the following four smells.

Dead metaclass is a design smell identifying a class not related to any other class in the metamodel, like `Affiliate`. This is analogous to *dead code* which is defined as a fragment of code that is no longer used [12]. This design smell may have a negative impact on the quality of the metamodel because it reduces its understandability and makes it more complex by introducing unusable and useless elements in the metamodel. Refactoring this smell consists of simply removing the class.

Classification by enumeration or by hierarchy is a design smell that concerns cases where we should use enumeration instead of an inheritance hierarchy of classes [29]. For instance, an `Employee` can be classified as a `FullTimeEmployee` or `PartTimeEmployee`. Depending on the use case, this could be considered as a design smell since the two sub-metaclasses do not add any new features. In that case, it could be more appropriate to represent the employment type as an enumeration inside `Employee` with these two values.

Concrete abstract metaclass is a design smell where super-classes are concrete instead of abstract. For instance, `Employee` should not be instantiated because an employee must be one of the two subtypes. To refactor this smell, we make the class abstract [8].

Duplicated features is another common smell, where the same feature is duplicated in several classes [29]. For example, some of the attributes are repeated in more than one class (e.g., `name` in `Employee` and `Client`). This design smell can be resolved by applying a *pull-up attribute* refactoring, if these classes have a common super-class (e.g., `interestRate`); otherwise, we should create a super-class that unifies the duplicated features (e.g., `name`).

Removing design smells is not a goal in itself, but aims at improving the quality of the metamodel. However, the quality attributes of metamodels are generally conflicting [2, 17]: improving one quality attribute may harm another one. For instance, improving the maintainability and abstraction of a metamodel by extracting several super-classes may adversely impact its understandability and complexity. Thus, removing all the smells or defining a static approach to remove smells is not necessarily the best solution, and may deteriorate the quality of the metamodel. Therefore, refactoring metamodels is a complex problem where we need to find a compromise between conflicting quality factors.

3 EVALUATION OF THE METAMODEL QUALITY

3.1 Overview

Our approach refactors metamodels to improve their quality through the removal of design smells. Figure 2 outlines the proposed approach. In ①, we reuse the library proposed in [8] which implements the detection and refactoring of some design smells. In ②, we implement a module to evaluate the metamodel at hand following a specific quality model. This module defines the objectives that will be used in the optimization process. These two blocks are internally used by our multi-objective optimization algorithm, in ③, to compute the objectives (i.e., quality attributes) and generate the target solutions. Our approach outputs the best sequence of refactorings that maximizes the quality of metamodel. Then, the modeler can select any solution from the list of recommendations, according to her context and preferences, and apply it to generate

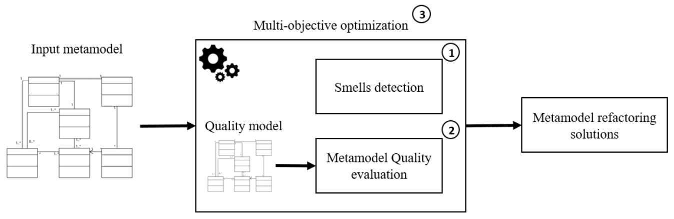
Multi-objective optimization
3
Input metamodel
1
Smells detection
2
Quality model
Metamodel Quality evaluation
Metamodel refactoring solutions

Figure 2: Proposed approach

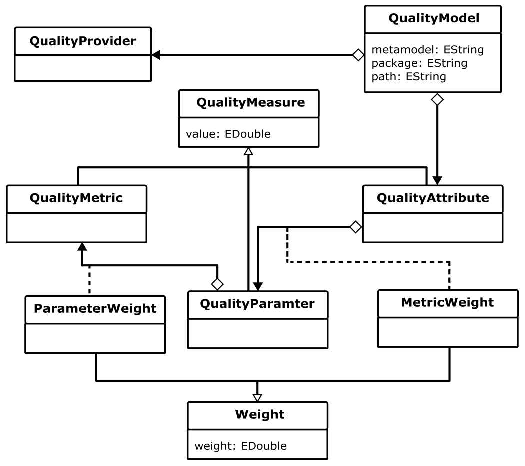

Figure 3: Metamodel for quality evaluation

the refactored version of the metamodel. We detail the different steps of our approach in what follows.

3.2 Promoting the quality of metamodels

We aim to generalize and automate our proposed approach to any quality framework. We define a quality metamodel, as shown in Figure 3. This metamodel is defined as a hierarchy of quality measures. It defines the quality of metamodels using three levels of abstraction with mappings from high to lower levels. The quality attributes are at the highest level, decomposed into quality parameters, in turn decomposed into quality metrics. Quality attributes are high-level concerns, such as understandability, maintainability, and extendibility. Although they are easy to interpret, they are not trivial to measure.

Ma et al.[22] presented a framework to evaluate the quality of metamodels. The provided quality model can be specified using our metamodel (Figure 3). Ma et al.[22] expressed quality attributes as a combination of lower-level metrics called quality parameters (e.g., size, coupling, inheritance), in turn computed using quality metrics such as number of classes, abstract classes, average depth of inheritance hierarchies, etc. Quality metrics are computed directly from the metamodel. We rely on the recursive combination of metrics and parameters, i.e., quality attributes, which are better suited for modelers to interpret. The quality provider is responsible for computing the different quality measures.

Our approach is generic: it supports any quality model as long as it conforms to the defined quality metamodel (Figure 3). In particular, this is the case of the quality model presented by Ma et al.[22]. The authors define a set of quality metrics: average number of abstractions (i.e., number of abstract meta-classes), number of concrete classes, average number of associations, average number of attributes per class, etc. They also define a mapping between quality metrics and parameters. For instance, coupling is defined as the sum of the average number of abstractions and the average number of associations. Finally, they define the quality attributes as a weighted sum of the quality parameters. Without loss of generality, this is the model we use in our experiments.

To evaluate the quality of a metamodel with respect to the defined quality model (i.e., compute the values of quality attributes), we used Edelta [7]. It is a textual domain-specific language (DSL) for metamodel refactoring and migration. Listing 1 shows a code snippet from our evaluation module. It computes two quality metrics: number of abstract classes and number of concrete classes. It is a general, reusable and extensible module that implements the different quality measures.

```
def computeNAC(EPackage p){
  return p.allEClasses.iterator.filter[c | c.abstract].size
}
def computeNCC(EPackage p){
  return p.allEClasses.iterator.filter[c | !c.abstract].size
}
```

Listing 1: Code snippet for metamodel quality evaluation

3.3 Design smell detection and refactoring

Design smells are structures in the design that indicate violations of fundamental design principles and negatively impacting design quality [31]. We used the smell detection library provided in [8]. It provides an implementation for detecting five metamodel smells: *Duplicated features*, *Dead metaclass*, *Redundant container relation*, *Classification by enumeration or by hierarchy* and *Concrete abstract class* (c.f. Section 2). Our approach is based on smell detection and correction. To remove these design smells, we also use the weaving model defined in [8]. It matches every bad smell with the corresponding refactoring operation that removes it.

4 SEARCH-BASED METAMODEL REFACTORING

4.1 Multi-objective optimization

The process of simultaneously optimizing a set of objective functions is called multi-objective optimization (MOO) or vector optimization [23]. It consists of the optimization of two or more conflicting objectives. Therefore, there is not a single solution that optimizes all the objectives, but rather, we can have a set of Pareto-optimal solutions [11]. A solution is called *non-dominated* or *Pareto-optimal* if there is no other solution that is better in all the objectives. These solutions are indiscernible and are all considered as good solutions.

Non-dominated Sorting Genetic Algorithm II (NSGA-II), which is used in this work, is a MOO evolutionary algorithm that is computationally fast. It is based on sorting the population on multiple fronts using a non-dominated rapid sorting approach. It also uses genetic operators (*crossover* and *mutation*) to generate new solutions and crowding distance to sort individuals. NSGA-II finds a representative set of non-dominated solutions that constitute a compromise between the predefined objectives.

In our approach, we define the problem as a MOO process where the goal is to find the best sequence of refactorings (i.e., the list of bad smells to be removed since each smell is associated with the refactoring operation that removes it) that optimizes our conflicting quality attributes objectives.

If the search space is small, i.e., small metamodel with few detected smells, we can just use an exhaustive search that consists of exploring the whole search space and testing all possible combinations then selecting the best ones. In this case, we are sure that the Pareto front we get is the globally optimal one. However, when it comes to large search space, it is difficult, if not impossible, to perform an exhaustive search considering the very high number of solutions to explore. In this case, a heuristic search reduces the search space by exploring fewer solutions and can find near-optimal solutions. More specifically, we use NSGA-II to explore the search space and find good refactoring solutions that maximize the quality of the input metamodel.

We first evaluate the size of the search space depending on the problem parameters, i.e., the size of the metamodel and the number of detected smells and we select the appropriate search method depending on the search space size. If an exhaustive search is doable in a reasonable time frame, then, we use it. Otherwise, we use the multi-objective genetic algorithm NSGA-II to wisely explore the space of potential solutions.

4.2 Problem formulation

We define the problem by expressing the objectives and the structure of target solutions. The quality attributes represent our objectives that we aim to maximize in our work. Additionally, we aim to maximize the removal of bad smells. Using many objectives expands the number of output solutions. Thus, we enable the user to specify the quality criteria she is interested in among five quality attributes: *Reusability*, *Understandability*, *Functionality*, *Extendibility*, and *Well-structured.* Taking user preferences into consideration allows us to reduce the number of suggestions and recommend relevant user-specific solutions.

We denote SM the set of all bad smells and RE the set of all refactorings. We define the mapping $refact : SM \longrightarrow RE$ such that $\forall sm \in SM, \exists re \in RE : refact(sm) = re$. A solution is represented as a vector $S_i = \{sm_i \in SM\}$ of bad smells to be removed. Each bad smell $sm_i \in S_i$ is associated with the corresponding refactoring $re_i = refact(sm_i)$ that removes it. Therefore, we can also consider our solution as a set of refactorings. The solutions might have different sizes where each solution is a subset of the complete list of smells initially detected from the input metamodel.

4.3 Generating new refactoring solutions

NSGA-II uses some genetic operators to combine solutions from the current iteration and generate new solutions. To this end, we define two operators: (i) crossover to produce new refactoring sets by exchanging subsets between two selected refactoring solutions and (ii) mutation that modifies the refactoring set of a given solution.

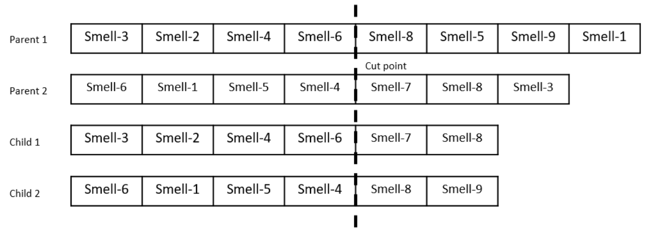

Figure 4: Adapted single-point crossover operator

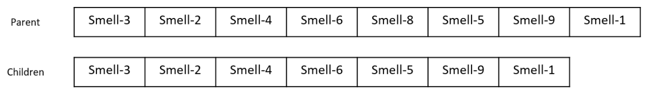

Figure 5: Adapted mutation operator

4.3.1 Crossover. We use single-point crossover and adapt it to our specific context. It consists of taking two refactoring solutions from the current population (parents), randomly choosing a cut point in the associated refactoring sets, then generating a new refactoring solution (child) that combines the left-side refactoring subset of the first solution with the right-side subset of the second solution relatively to this point. A second refactoring solution (child) is created by combining the remaining subsets.

For our specific problem, one problem that may occur is having a new child solution with duplicate instances of the same bad smells. This case occurs when the left side of the first parent contains the same bad smell instance as the right side of the second parent. To overcome this problem, we adapt the crossover operator by removing duplicates in the new child if detected. Figure 4 illustrates an example of crossover operation applied on two parent refactoring solutions. We form one child refactoring solution by combining the subsets of the two parent solutions as described previously. Then, we check for duplicates. Since the same bad smell instance, *smell-3*, appears twice, we only retain the first instance. Similarly, the second refactoring solution produced consists only of one instance of the smells combined from the parents.

4.3.2 Mutation. For each of the bad smells in the solution, we decide with a certain probability (i.e., mutation rate) whether to delete or not the bad smell instance. This means that the produced refactoring solution is composed of a subset of the parent smells instances. Figure 5 shows an example where the application of the mutation operator on the parent solution led to removing *Smell-8* and keeping other elements.

Since mutation removes smells, we may end up with a smell that will never appear in a solution of the Pareto front. We mitigate this risk by diversifying our initial population with different smells. Moreover, in NSGA-II, the best half of the solutions is systematically added to the next generation of solutions. This significantly lowers the risk of not encountering a smell in the Pareto.

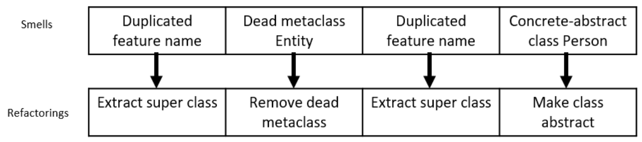

Figure 6: Example of mapping design smells to their corresponding refactoring operations

4.4 Evaluating refactoring solutions

We consider a solution to be valid if it is applicable: all refactorings corresponding to the bad smells in the solution are not conflicting. The application of one refactoring operation may invalidate another one. For instance, extracting a duplicate feature from a dead metaclass, that is already deleted, will produce an error when applying the solution. We overcome this problem using a trial-and-error approach. We apply the solution and we check if it is valid. If it is invalid, we eliminate it from the list of solutions. In the optimization phase, we only retain valid solutions that are applicable to avoid misleading the user.

A solution is a set of smell instances to be removed. To evaluate a solution, we translate it to a list of corresponding refactorings (see for example Figure 6). This is done by replacing each smell with its corresponding refactoring operation that allows to remove it. Then, we apply these refactorings on the input metamodel to generate the refactored metamodel. Finally, we evaluate the quality of the resulting metamodel by computing its quality attributes using our evaluation framework (Section 3.2).

We define two objective functions to optimize for the MOO in NSGA-II. They take as input the metamodel $m \in M$ (where M is the set of all models) and a solution S_i as defined in Section 4.2. One objective is to maximize all the quality attribute values of the metamodel: $\max\{qual(m, qa), \forall qa \in QA\}$ where $qual : M \times QA \rightarrow \mathbb{R}$ is computed as described in Section 3. The second objective is to minimize the number of smells remaining in the metamodel: $\min\{|S_i| : sm_i \in smell(m)\}$, where $smell : M \rightarrow \mathcal{P}(SM)$ returns all the bad design smells in the metamodel. A solution dominates another if it has better objective values (better quality values and more smells to remove). Otherwise, they are non-dominated. This means that they are indiscernible since each solution has better values for some of the objectives.

4.5 Metamodel refactoring recommendation

As mentioned earlier, NSGA-II produces a Pareto front of non-dominated solutions that are equally good since each solution improves some objectives (i.e., quality attributes) with respect to other solutions. Therefore, it is up to the user to select the most appropriate solution that suits her preferences and her specific context. Table 1 shows an example of two non-dominated solutions executed on one of the metamodels used in our validation (Customer-Relationship management — CRM). We cannot discriminate between these two solutions since each of them has better values for some of the objectives. Solution 1 has better values compared to Solution 2 in *Understandability* and *Well-structured* objectives, but worse values in *Reusability*, *Functionality*, and *Extendibility*.

Table 1: Example of non-dominated solutions: objective values

	Solution 1	**Solution 2**
Reusability	6.2	**6.5**
Understandability	**3.8**	3.1
Functionality	4.8	**4.9**
Extendibility	2.1	**2.6**
Well-structured	**3.9**	3

Therefore, choosing the appropriate solution depends on which quality factors the user favors according to her specific context.

We give further visual assistance to the user by presenting charts reporting the values of each quality attribute.

5 VALIDATION

We conducted a set of experiments to evaluate our approach quantitatively and qualitatively. In this section, we present our research questions followed by the experimental setting. Then, we present and discuss the results.

5.1 Research questions

To validate our approach, we define four main research questions:

- **RQ0:** *Are the results of our approach attributable to the search strategy or to the number of explored solutions?* This is a sanity check to ensure that our approach does not find the best solutions by chance, but because of the method used.
- **RQ1:** *Given a quality model, does our search-based refactoring approach improve the quality of the metamodel better than not applying any refactoring or removing all design smells?* We compare the quality of the metamodels resulting from three methods: the initial version of the metamodel with all the design smells, the refactored metamodel with all smells removed applying the approach in [8], and applying the refactorings with our search-based approach.
- **RQ2:** *How does our approach perform in practice compared to other alternatives with respect to independent user assessment of understandability?* We assess which approach specifically improves the understandability quality attribute of given metamodels in practice. Thus, we rely on a user study to evaluate the metamodels resulting from our approach, those refactored using the approach in [8], and the initial metamodels.
- **RQ3:** *How does our approach perform in practice compared to other alternatives with respect to independent user assessment of extendibility?* This research question is similar to the previous one, but we focus on the extendibility quality attribute instead.

5.2 Experimental setting

To address the different research questions, we conducted two sets of experiments: an empirical study based on a dataset of metamodels for **RQ0** and **RQ1**, and a user study for **RQ2** and **RQ3**.

5.2.1 Empirical study based on the quality model.

To answer **RQ0** and **RQ1**, we use a dataset of 10 metamodels selected from an online Ecore dataset[1]. Table 2 presents the characteristics of each metamodel: number of classes in the metamodel, total number of attributes, number of associations, number of inheritance relations, and the number of bad design smells detected.

Table 2: Characteristics of the metamodels

Metamodel	Classes	Attr.	Assoc.	Inherit.	Smells
CRM	11	6	8	5	6
DBLP	17	51	17	8	13
BibTeX	29	55	4	48	17
Ant	51	95	28	40	25
Maven	8	28	6	2	3
jPQL	49	24	42	35	20
HTML	59	98	14	42	38
SQL	92	45	119	42	42
WikiML	29	10	22	24	12
MongoSQL	23	21	6	18	8

During the experiments, we used the quality model presented in [22]. Based on some recommended configurations in the literature [18], we set the hyper-parameters of NSGA-II for the population size to 50, the crossover and mutation probabilities to 0.8, and the maximum number of generations to 400.

To answer **RQ0**, we implemented a random search algorithm with a uniform distribution. It is a direct optimization method that does not require a search strategy to generate solutions. We use a uniform distribution to build our solution from the list of detected design smells. The probability of a smell to be removed from an input metamodel is 0.5. We run a number of fixed iterations. For each iteration, we build a new random solution and only keep the best solution that optimizes the objective function. We compare the results for the same number of generated solutions using random search and our search-based approach. This allows us to prove whether the achieved results are attributable to the search strategy or to the number of explored solutions.

To answer **RQ1**, we consider the initial metamodel with all the design smells (*Initial*), the refactored metamodel with all smells removed by applying the approach in [8] (*All*), and the metamodel refactored by applying the refactorings with our search-based approach (*MOO*). For each metamodel in Table 2, we compare the three versions with respect to the same quality model [22] by evaluating the quality attributes of each solution.

5.2.2 User study targetting specific quality attributes.
The results obtained to answer the previous research questions highly depend on the quality model provided, since it is what guides the optimization process in MOO. This may introduce a bias in the evaluation results. To assess a neutral and independent evaluation from the quality model, we conducted a user study where we asked participants to perform tasks related to specific quality attributes. We use the results of the user study to answer **RQ2** and **RQ3** from a practical perspective.

[1]https://doi.org/10.5281/zenodo.2585456

Unlike the quality-model based study, we focus on improving two specific quality attributes, namely understandability and extendibility. We selected three metamodels: a *bank* management system, an online store system *estore*, and a customer-relation management system *CRM* (Table 2). For each metamodel, we consider the three versions: *Initial*, *All*, and *MOO* produced using the methods above. For MOO, we employ the same configuration of hyper-parameters as in the first study, but optimize only three objectives: understandability, extendibility, and maximizing the number of removed smells. Then, we randomly select a solution from the list of recommendations.

To enroll participants in the study, we sent a call for volunteers requiring knowledge in UML class diagrams and MDE. We followed a convenience sampling method, by sending invitations to research groups in our network. Nine participants responded positively, who are researchers (Ph.D. students, postodoctoral fellows, or professors) with good expertise in metamodeling. We designed the experiment as shown in Table 3. We divided the participants into three groups of three, such that each group is treated with exactly one metamodel per version.

Table 3: Design of the user study

Metamodel / Version	Initial	All	MOO
bank	Group 1	Group 2	Group 3
estore	Group 2	Group 3	Group 1
CRM	Group 3	Group 1	Group 2

For each metamodel, we designed two questions per quality attribute with different levels of difficulty. For understandability (**RQ2**), we asked participants about some information encoded in the metamodel to evaluate the level of comprehension of a given version of the metamodel. For example, an easy question we asked for the bank metamodel (Figure 1 shows the *Initial* version) is *"What information does the bank have on saving accounts?"*. For extendibility (**RQ3**), we asked participants to verbally explain the changes they would perform to extend the metamodel with a new requirement. For example, a difficult question we asked for the *estore* metamodel is *"We want to extend the metamodel so that workers could supply services to clients who are able to book appointments with workers. An appointment has a time and a description. What are the required changes?"*. We collected the answers by taking notes. We kept these notes anonymous by assigning the answers to IDs instead of participants' names.

We allotted each participant 15 minutes in a one-on-one Zoom session. Through screensharing, we presented a version of a metamodel and the questions corresponding to that metamodel. We measured the time they took to answer a question (in seconds) and we recorded their responses to evaluate their correctness. All the material for this experiment is available online[2] for replication.

5.3 Results

5.3.1 ***Results for RQ0.*** We executed random search and our approach on the dataset of metamodels (Table 2). We compared the

[2]https://github.com/OussamaSghaier/SAC2022

non-dominated solutions generated by our approach to the best solution produced by random search algorithm. Figure 7 shows the results of our experiments on the CRM metamodel. We randomly selected two solutions (MOO-1 and MOO-2) from the recommended solutions generated by our approach. We see that the MOO solutions dominate the solution generated by random search in all the objectives. Our results show that our approach produces solutions that dominate the solution generated by random search in all the objectives for all metamodels. Therefore, we conclude that the obtained results are attributable to the search strategy and not to the number of explored solutions.

5.3.2 ***Results for RQ1.*** We obtain similar results when comparing our search-based approach to the solution produced by the approach presented in [8], as shown in Figure 7. Our approach results in better quality scores than the previous approach. Therefore, we conclude that removing all design smells does not always guarantee an improvement of overall quality. Instead, targetting refactoring operations that remove specific design smells yields a better metamodel quality. Interestingly, we note that removing all design smells may even worsen some quality attributes, in particular extendibility. For example, we observe that the *Extendibility*(*Initial*) > *Extendibility*(*ALL*) for the CRM metamodel.

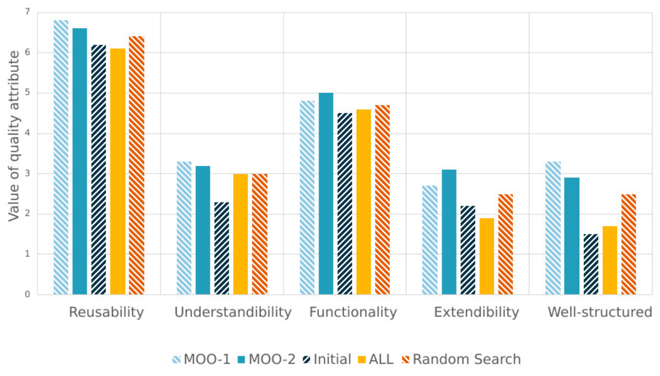

Figure 7: Comparison of the objectives values (i.e., quality attributes) for the best solution generated by the different methods on the CRM metamodel

5.3.3 ***Results for RQ2.*** For each question during the user study, we calculated the precision and recall of the participants' answers we collected as follows:

$$precision = \frac{\#\ relevant\ elements\ included\ in\ answer}{\#\ elements\ included\ in\ answer}$$

$$recall = \frac{\#\ relevant\ elements\ included\ in\ answer}{\#\ relevant\ elements}$$

We verified the normality of the results (i.e., time, precision, and recall) using *Kolmogorov-Smirnov* test. We obtained a significant $p - value < 0.001$. This means that there is a significant difference between the distribution of our results and the normal distribution. Therefore, we performed Mann-Whitney test on the time and score metrics we collected for the understandability questions. This is a non-parametric test of the null hypothesis that the observed differences between two independent variables were obtained by chance (i.e., null hypothesis) or if they are statistically significant ($p - value < 0.05$).

Table 4 describes the statistical results related to understandability of the different approaches. The statistically significant results are displayed in bold, meaning they are generalizable beyond our sample within similar settings. Time differences are significant when MOO is compared with the other approaches, but not significant for score precision and recall when comparing our approach with the one in [8].

Table 4: Pairwise statistical significance of time, precision and recall between the different versions of the metamodels

Quality attribute	Metamodel	Mann Whitney p-value		
		Time	Precision	Recall
Understandability	Initial - All	0.424	0.719	0.462
	Initial - MOO	**0.003**	0.239	0.111
	All - MOO	**0.009**	0.079	0.424
Extendibility	Initial - All	0.372	0.767	0.563
	Initial - MOO	**0.010**	**0.001**	**0.016**
	All - MOO	**0.000**	**0.000**	**0.000**

Figure 8 presents the precision and recall of the results for each answer related to understandability using the different approaches. It is clear that our approach provides better values of precision and recall. This is true both in terms of centrality and dispersion. As a baseline, more than 85% of the understandability-related responses were correct with respect to the initial metamodel. However, there is a lot of variability among the participants, some are clearly not able to correctly answer understandability questions. This is due to the large number of bad smells present in the initial metamodel. Removing all bad smells yields a perfect precision and improves the recall. However, there are still low values. This is explained because removing all the smells may increase the size of the metamodel, making it less understandable. In contrast, when presenting the metamodels produced by our approach, almost all participants scored perfectly. Furthermore, all values are contained in the 78% to 100% interval. This means that the understandability by the developers is uniform and better when metamodels are refactored using MOO.

The time measured to answer understandability questions corroborates these results. As depicted in Figure 9, it takes the participants significantly less time to answer the questions correctly. Interestingly, in general, it took them more time to understand metamodels where all smells are removed. The *All* approach introduces many new elements in the metamodel, which makes it more complex and increases the time required to understand it.

Figure 10 depicts an excerpt from the different versions of Bank Management System metamodel (i.e., initial metamodel, metamodel generated using ALL approach and metamodel generated using our MOO approach). The initial version of the metamodel contains some instances of bad smells: *duplicated feature* (e.g., `id`, `interestRate`, `amount`), *dead metaclass* (e.g., `Affiliate`). The *All* approach removes all these bad smells by applying the corresponding refactoring operations: `pull-up-feature(interestRate)`, `remove-dead-metaclass(`

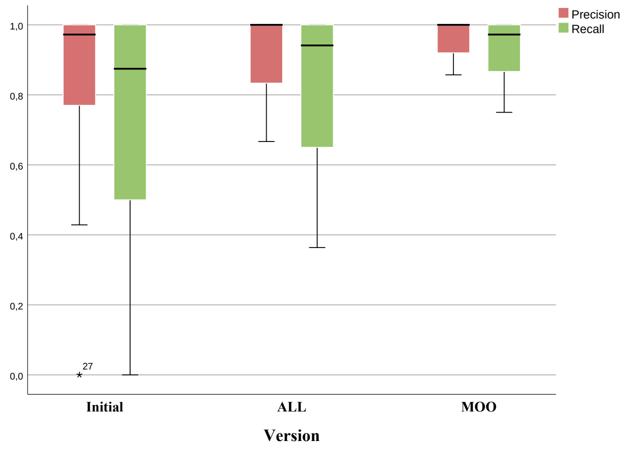

Figure 8: Precision and recall of the results in terms of understandability

`Affiliate)`, `create-super-metaclass(id)`, `create-super-metaclass(amount)`. This leads to a more complex design by introducing several meta-classes with few features compared to our MOO approach that removes a subset of the detected smells. This harms significantly the comprehension of the metamodel and increases the time required to understand it.

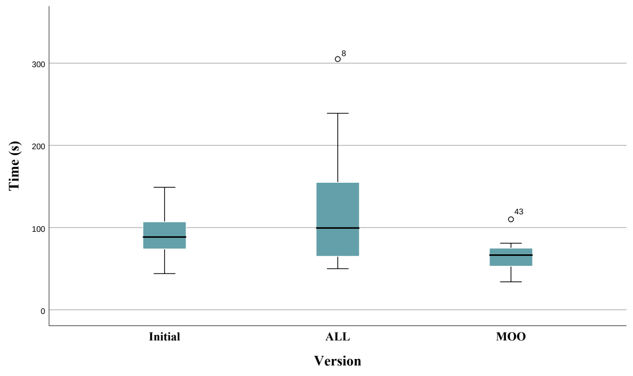

Figure 9: Time measurements in terms of understandability

5.3.4 ***Results for RQ3.*** In terms of extendibility, Table 4 shows that the results obtained are significant for time, score precision and recall when MOO is compared with the other two approaches. From Figure 11, we can see that our approach outperforms the other alternatives as it was the case for understandability. Furthermore, we note a high dispersion for the *Initial* and *All* metamodels. As a baseline, the participants had significant difficulties correctly extending the *Initial* metamodel. However, removing all smells from a metamodel does not help in extending it neither. The amount of changes resulting from removing all smells makes the metamodel even more complex to extend. For example, we noted that some participants forgot to include key elements in their responses.

With no surprise, the participants required less time to answer the extendibility questions when provided a metamodel produced by *MOO* compared to the other two (see Figure 12).

We investigated the correlation between the time and correctness of the responses by measuring the *Pearson correlation coefficient*. Table 5 shows that the time and the correctness scores (i.e., precision and recall) are inversely correlated. This correlation is weak (≈ -0.3) but statistically significant ($<= 0.01$). This means that participants who took more time to answer questions were more likely to give wrong answers. However, this relationship between the correctness of the responses and the time allotted by the participants to answer questions is minimal. We believe that correctness depends more on the quality of the design.

Table 5: Pearson correlation between Time and Precision / Recall

		Precision	**Recall**
Time	**Pearson correlation**	−0.372	−0.305
	Significance	0	0.01

The main outcome of this user study is that our approach to refactoring metamodels significantly improves the comprehension and the extension of metamodels.

6 THREATS TO VALIDITY

The evaluation of our approach revealed compelling evidence that explicitly addressing quality concerns in metamodel refactoring is more beneficial than just removing all smells. These results must, however, be read in the context of our evaluation setup. In this section, we discuss different threats that may limit the validity of the study presented in Section 5.

From the **internal validity** perspective, we identified two threats, maturation and confounding. Firstly, we assigned the metamodels/refactoring approach in a way that no participant will see the same meta-model twice. Moreover, the metamodels were presented in a random order to the participants to avoid maturation threats caused by learning, fatigue, or boredom, in addition to limit the duration of the evaluation sessions. For the confounding threat, we ensured that the participants have the sufficient knowledge in metamodeling and assigned them randomly to the groups. Furthermore, as each participant experimented with all the refactoring options, we do believe that the results are attributable to the refactoring approach and not to the knowledge of participants or their intrinsic ability to perform the tasks quickly.

For the **external validity**, we identified some threats. The first relates to the metamodels used in the evaluation of our approach. These have average sizes. Although, bigger metamodels would be more representative, we decided to avoid them to limit the effort of the participants. Also, our choice of metamodels with different characteristics may help mitigating this threat. The second threat concerns the choice of the quality model, used to evaluate the different approaches in the first experiments. This is part of our optimization process. This makes the results of the first experiment biased towards our approach. Therefore, we conducted user-based

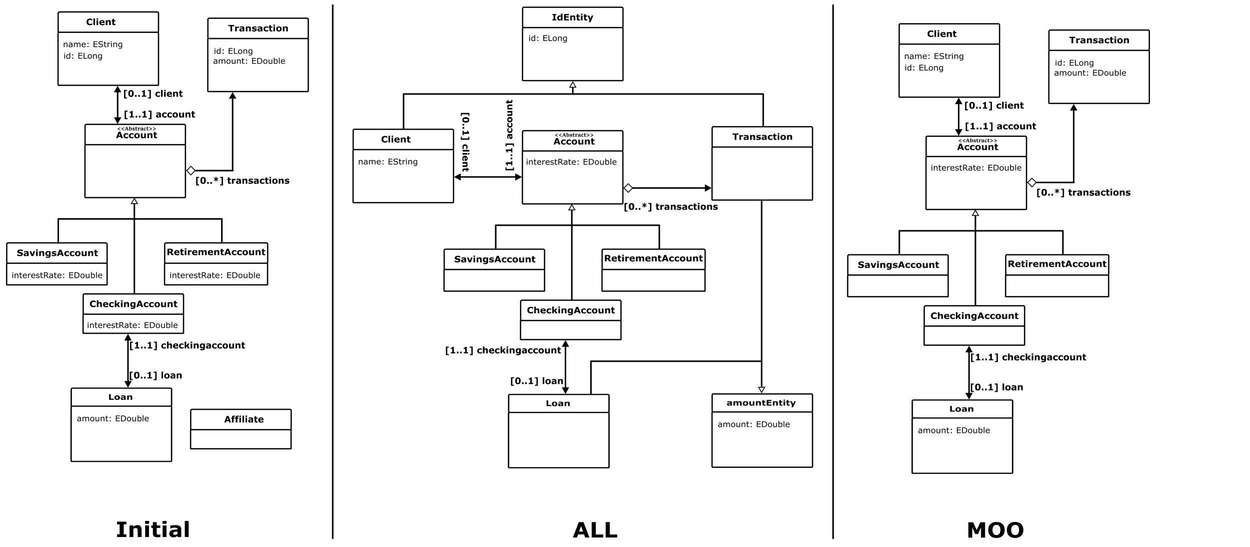

Figure 10: An excerpt of Bank Management System metamodel: different versions of the metamodel (i.e., initial metamodel, metamodel generated using the ALL approach and metamodel generated using our MOO approach)

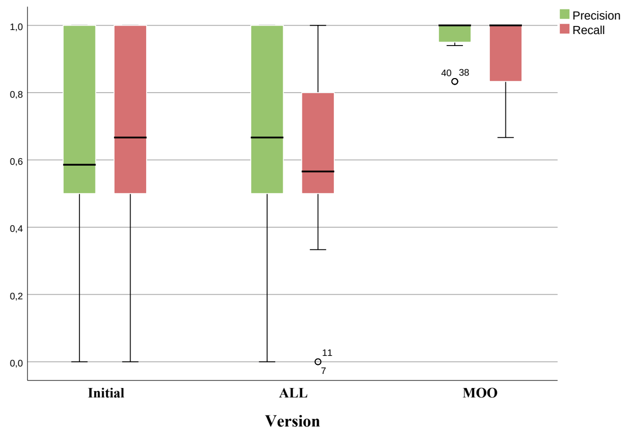

Figure 11: Precision and recall of the results in terms of extendibility

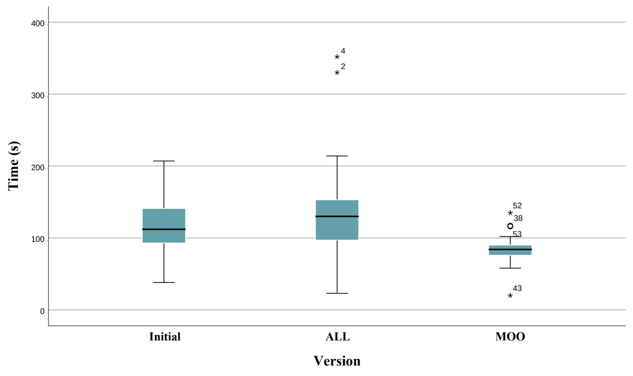

Figure 12: Time measurements in terms of extendibility

experiments to evaluate the effectiveness of our approach and its performance independently from the used quality model. Another threat to external validity refers to the size of the subjects involved. We mitigated this threat by assessing the statistical significance of the results rather than just reporting the sample results. Still, it is necessary to replicate this study with more subjects and metamodel sets, to draw final conclusions.

For the **construction validity**, we prevented the mono-method bias by using three measures of the performance of the respective refactoring alternatives: time, precision and recall.

7 RELATED WORK

In the literature, many approaches have investigated the automation of model or metamodel refactoring using formal methods (e.g., [15]), model transformations (e.g., [33]), learning process from preexisting examples (e.g., [19]), etc..

Several approaches were proposed to learn model transformations from examples. In [14], the authors use a search-based approach to generate the best sequence of refactorings based on textual and structural similarity with a set of provided examples. In [5], the authors propose a process to learn complex model transformations by considering three common requirements: element context and state dependencies and complex value derivation. Kessentini et al.[19] use a set of transformation examples to derive a target model from a source model. The authors explore different transformation possibilities evaluated based on their conformance with the examples at hand. A similar work was proposed in [25] that relies on genetic programming to learn model transformation rules guided by the conformance with the provided examples.

EMF Refactor tool was introduced in [3, 4]. It supports metrics reporting, smells detection and resolution for models based on Eclipse Modeling Framework (EMF) [28]. Other approaches consist of deriving endogenous and in-place model transformations. Reimann et al.[27] present a generic refactoring framework based on EMF to model refactorings for different modeling and metamodeling languages. These generic refactorings can be reused for different languages only by providing a mapping. Based on the defined mapping, a generic transformation is executed to restructure the models. In [10, 30], the authors derive model transformations by analyzing user editing actions when refactoring models.

In [15], the authors present a constraint-based approach to refactor models. The semantics of Unified Modeling Language (UML) were defined as well-formedness rules (WFRs) to guide the refactoring process for a set of connected models. The authors check these WFRs on transformed models. If not satisfied, they solve the failed constraints by computing additional model changes required to have a valid and semantic-preserving refactoring transformation.

Bettini et al.[8] propose a quality-driven framework for detecting and resolving metamodel smells. The authors define static mapping between bad smells and quality attributes by associating each smell to the set of quality attributes it affects. Based on quality requirements, the associated smells are identified and removed using refactoring operations.

Most of the cited works do not rely on using the conflicting quality factors to guide the model refactoring process, yet its main goal is improving the design quality.

8 CONCLUSION

In this paper, we presented an approach to recommend refactorings for metamodels using multi-objective heuristic search. The optimization process is driven by the improvement of some quality attributes selected by the user. To evaluate the effectiveness of our approach, we conducted a set of experiments using a dataset of metamodels and relying on participants familiar with metamodeling. The results show that our approach succeeds in improving the quality of an input metamodel and outperforms other proposed approaches.

As part of future work, we plan to improve our proposed framework to limit the number of proposed solutions by keeping the most relevant. We intend to explore running a second optimization phase to reduce the size of solutions space based on the user preferences.

REFERENCES

[1] Marwen Abbes, Foutse Khomh, Yann-Gael Gueheneuc, and Giuliano Antoniol. 2011. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *15Th european conf. on software maintenance and reengineering*. 181–190.

[2] Bill Andreopoulos. 2004. Satisficing the Conflicting Software Qualities of Maintainability and Performance at the Source Code Level.. In *WER*. 176–188.

[3] Thorsten Arendt and Gabriele Taentzer. 2012. Integration of smells and refactorings within the eclipse modeling framework. In *Proceedings of the Fifth Workshop on Refactoring Tools*. ACM, 8–15.

[4] Thorsten Arendt and Gabriele Taentzer. 2013. A tool environment for quality assurance based on the Eclipse Modeling Framework. *Automated Software Engineering* (2013), 141–184.

[5] Islem Baki and Houari Sahraoui. 2016. Multi-step learning and adaptive search for learning complex model transformations from examples. *ACM Transactions on Software Engineering and Methodology* (2016), 1–37.

[6] Manuel F Bertoa and Antonio Vallecillo. 2010. Quality attributes for software metamodels. *Málaga, Spain* (2010).

[7] Lorenzo Bettini, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2017. Edelta: An Approach for Defining and Applying Reusable Metamodel Refactorings.. In *MODELS*. 71–80.

[8] Lorenzo Bettini, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. 2019. Quality-Driven Detection and Resolution of Metamodel Smells. *IEEE Access* (2019), 16364–16376.

[9] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. Model-driven software engineering in practice. *Synthesis lectures on software engineering* (2017), 1–207.

[10] Petra Broschy, Philip Langer, Martina Seidl, and Manuel Wimmer. 2009. Towards end-user adaptable model versioning: The by-example operation recorder. In *ICSE Workshop on Comparison and Versioning of Software Models*. 55–60.

[11] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and Tanaka Meyarivan. 2000. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In *Int. conf. on parallel problem solving from nature*. 849–858.

[12] Saumya K Debray, William Evans, Robert Muth, and Bjorn De Sutter. 2000. Compiler techniques for code compaction. *ACM Transactions on Programming languages and Systems* (2000), 378–415.

[13] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[14] Adnane Ghannem, Marouane Kessentini, Mohammad Salah Hamdi, and Ghizlane El Boussaidi. 2018. Model refactoring by example: A multi-objective search based software engineering approach. *Journal of Software: Evolution and Process* (2018).

[15] Rohit Gheyi, Tiago Massoni, and Paulo Borba. 2005. A rigorous approach for proving model refactorings. In *20th IEEE/ACM international Conf. on Automated software engineering*. 372–375.

[16] Aakriti Gupta and Shreta Sharma. 2015. Software Maintenance: Challenges and Issues. *Issues* (2015), 23–25.

[17] Neil B Harrison and Paris Avgeriou. 2007. Leveraging architecture patterns to satisfy quality attributes. In *European conf. on software architecture*. 263–270.

[18] Ahmad Hassanat, Khalid Almohammadi, Esra' Alkafaween, Eman Abunawas, Awni Hammouri, and VB Prasath. 2019. Choosing mutation and crossover ratios for genetic algorithms—a review with a new dynamic approach. *Information* 10, 12 (2019), 390.

[19] Marouane Kessentini, Houari Sahraoui, Mounir Boukadoum, and Omar Ben Omar. 2012. Search-based model transformation by example. *Software & Systems Modeling* (2012), 209–226.

[20] Foutse Khomh, Massimiliano Di Penta, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2012. An exploratory study of the impact of antipatterns on class change-and fault-proneness. *Empirical Software Engineering* (2012), 243–275.

[21] Jesús J López-Fernández, Esther Guerra, and Juan De Lara. 2014. Assessing the Quality of Meta-models.. In *MoDeVVa@ MoDELS*. 3–12.

[22] Zhiyi Ma, Xiao He, and Chao Liu. 2013. Assessing the quality of metamodels. *Frontiers of Computer Science* (2013), 558–570.

[23] R Timothy Marler and Jasbir S Arora. 2004. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization* (2004), 369–395.

[24] Maddeh Mohamed, Mohamed Romdhani, and Khaled Ghédira. 2009. Classification of model refactoring approaches. *Journal of Object Technology* 8, 6 (2009), 121–126.

[25] Chihab eddine Mokaddem, Houari Sahraoui, and Eugene Syriani. 2018. Recommending model refactoring rules from refactoring examples. In *21th ACM/IEEE Int. Conf. on Model Driven Engineering Languages and Systems*. 257–266.

[26] Haris Mumtaz, Mohammad Alshayeb, Sajjad Mahmood, and Mahmood Niazi. 2019. A survey on UML model smells detection techniques for software refactoring. *Journal of Software: Evolution and Process* (2019), e2154.

[27] Jan Reimann, Mirko Seifert, and Uwe Aßmann. 2010. Role-based generic model refactoring. In *International Conf. on Model Driven Engineering Languages and Systems*. 78–92.

[28] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: eclipse modeling framework*. Pearson Education.

[29] Misha Strittmatter, Georg Hinkel, Michael Langhammer, Reiner Jung, and Robert Heinrich. 2016. Challenges in the evolution of metamodels: Smells and antipatterns of a historically-grown metamodel. (2016).

[30] Yu Sun, Jeff Gray, and Jules White. 2011. MT-Scribe: an end-user approach to automate software model evolution. In *33rd international conf. on software engineering*. 980–982.

[31] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. 2014. *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann.

[32] Aiko Yamashita and Leon Moonen. 2013. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *35th International Conf. on Software Engineering*. 682–691.

[33] Jing Zhang, Yuehua Lin, and Jeff Gray. 2005. Generic and domain-specific model refactoring using a model transformation engine. In *Model-driven Software Development*. 199–217.