Towards automating the construction of recommender systems for low-code development platforms

Lissette Almonte
lissette.almonte@uam.es
Universidad Autónoma de Madrid
Madrid, Spain

Iván Cantador
ivan.cantador@uam.es
Universidad Autónoma de Madrid
Madrid, Spain

Esther Guerra
esther.guerra@uam.es
Universidad Autónoma de Madrid
Madrid, Spain

Juan de Lara
juan.delara@uam.es
Universidad Autónoma de Madrid
Madrid, Spain

ABSTRACT

Low-code development platforms allow users with a low technical background to build complete software solutions, typically by means of graphical user interfaces, diagrams or declarative languages. In these platforms, recommender systems play an important role as they can provide users with relevant, personalised suggestions generated according to previously developed software solutions. However, developing recommender systems requires a high investment of time as it implies the selection and implementation of a suitable recommendation method, its configuration for the problem and domain at hand, and its evaluation to assess the accuracy of its recommendations.

To alleviate these problems, in this paper, we present the first steps towards a generic model-driven framework capable of generating ad-hoc, task-oriented recommender systems for their integration on low-code platforms. As a proof of concept, we present some preliminary results obtained from an offline evaluation of our framework on three datasets of class diagrams. The results show that the proposed framework is capable of providing relevant recommendations in the given context.

CCS CONCEPTS

• **Software and its engineering → Development frameworks and environments**.

KEYWORDS

Low-code platform, Model-driven engineering, Recommender system, Domain-specific languages

ACM Reference Format:
Lissette Almonte, Iván Cantador, Esther Guerra, and Juan de Lara. 2020. Towards automating the construction of recommender systems for low-code development platforms. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20 Companion), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3417990.3420200

MODELS '20 Companion, October 18–23, 2020, Virtual Event, Canada

ACM ISBN 978-1-4503-8135-2/20/10…$15.00
https://doi.org/10.1145/3417990.3420200

1 INTRODUCTION

Model-Driven Engineering (MDE) is a widely used approach for software development employing high-level formal abstraction [8]. This approach uses models as the key development artefact. In MDE, models can be used to specify, analyse, test, simulate, execute, generate code and maintain systems. Models are frequently defined using Domain-Specific Languages (DSLs) that provide primitives and concepts representing the abstractions of a domain in an accurate manner [6, 34]. A meta-model, which is itself a model, describes the abstract syntax of a DSL. Hence, MDE is considered an efficient way to improve the quality of software by automating repetitive, error-prone or time-consuming tasks [8].

On the other hand, Low-Code Development Platforms (LCDPs) are software development platforms on the Cloud [32]. They follow a Platform-as-a-Service (PaaS) model whereby users can build fully operational applications via graphical interfaces, diagrams and declarative languages. LCDPs provide users with a fully managed environment for the entire application life cycle. The users of LCDPs – called *citizen developers* – typically lack a background in programming. Hence, it is important that LCDPs are able to integrate useful, easy-to-use mechanisms to assist these users in their development tasks. Recommender systems are one of such mechanisms.

In Software Engineering (SE), modern Integrated Development Environments (IDEs) offer plenty of functionalities to help developers being more efficient, such as intelligent auto-completion or context-aware quick fixes [7]. Recently, the SE community has been proposing tools to support SE tasks in a personalised way. Some of these tools incorporate recommender systems to help developers finding information and making decisions [23]. In SE, some examples of this include APIs usage pattern recommendation [19], the recommendation of source code [4] and debugging-related information [5] useful to the software engineer.

In this paper, our goal is to enhance LCDPs with recommender systems targeted to citizen developers, akin to those found in modern IDEs. These systems would exploit the knowledge captured by applications previously defined within the LCDP, in order to

provide personalised suggestions on how to complete new applications (e.g., adding missing attributes, or incorporating new concepts formerly used in similar solutions). The recommendations may be available using a range of comprehensible means for citizen developers, such as indicators in the diagrams, or chatbots interacted by natural language. In this context, the challenge from the LCDP developer's point of view is to devise mechanisms that simplify the creation of such recommender systems. This is a non-trivial task that encompasses the selection of a recommendation method suitable for the target problem, its adaptation to the concepts managed by the LCDP, and its training and evaluation until suggestions are proved to be reliable enough.

To address this task, this paper proposes an MDE solution to automate the generation of recommender systems for LCDPs. It is based on a DSL to describe the different aspects of recommender systems, including a description of the recommended items and their features, the users of the recommendations, the recommendation method, and the evaluation procedure. As a proof of concept, we apply our solution to the development of a recommender system of missing class diagrams elements (attributes, methods and superclasses) on three existing class diagram datasets. With this case study, we aim to answer the following research questions: **RQ*1*** *Can a recommender system help in class modelling tasks?*, **RQ*2*** *Which recommendation method of relevant attributes, methods and superclasses has the best performance?*, **RQ*3*** *Can hybrid approaches be beneficial for the recommendation of attributes, methods and superclasses?*, **RQ*4*** *Which method performs better when considering user and item coverage in the recommendation of relevant attributes, methods and superclasses?*.

This paper is organised as follows. In Section 2, we present some background on recommender systems. Section 3 introduces a motivating example and the architecture of our proposed solution. Section 4 describes the main components of our proposal. Then, Section 5 shows our proof-of-concept evaluation. Finally, we position our work with respect to the state-of-the-art in Section 6 and present the conclusions and directions for future work in Section 7.

2 RECOMMENDER SYSTEMS

Recommender Systems (RSs) emerged in the mid-90s as an independent research field from areas such as cognitive science, approximation theory and information retrieval [1]. RSs are software tools and techniques that suggest items considered relevant for a particular user. "Item" is the prevalent word to refer to what the system recommends, e.g., the products to buy on an online retail store, or the songs to listen on a music streaming service provider platform. These systems support individuals to evaluate an overwhelming amount of item options [22]. For this purpose, RSs may exploit item characterizations based on a range of item features (e.g., the genre in a movie recommender) [1].

RSs can be classified into the following three broad categories according to how the recommendations are made: *content-based*, where users are recommended items similar to the ones they preferred before; *collaborative filtering*, where users are recommended items that other people with similar preferences like; and *hybrid*, which combines the previous two techniques to avoid the limitations of the content-based and collaborative methods. Specifically, content-based approaches tend to have limited content analysis, and present limitations in user preference sparsity and user and item cold-start situations. A user cold-start situation happens when the user has only rated a reduced number of items, in which case, the RS cannot profoundly understand the user preferences and provide accurate recommendations [1]. The item cold-start situation is similar, but for items.

Another way to classify RSs is based on the recommendation output. This can be either an estimation of user preference values (usually expressed in the form of numeric ratings) for items, or the generation of an ordered (ranked) list of the most relevant items. To measure the RS performance, there are different metrics for each type of approach. Some metrics are based on the rating prediction error (e.g., MAE, RMSE), and others measure the item ranking quality (e.g., precision, recall, nDCG, MRR) [13].

In general, an RS is built and operates with matrices that capture the user and item attributes and interactions. As an example, Figure 1(a) shows a *user-item matrix*, where I_0–I_3 represent items, U_0–U_3 represent users, and each cell *R(u,i)* of the matrix contains the rating given by the user u to the item i (a dash if the user has not rated the item) [22, 31]. Similarly, Figure 1(b) shows an example of an *item-feature matrix*. In this case, I_0–I_3 are items, f_0–f_3 are item features, and each cell is set to 1 if the item has the feature, and to 0 otherwise.

	I_0	I_1	I_2	I_3
U_0	2	-	-	1
U_1	1	5	5	2
U_2	5	-	3	2
U_3	4	5	4	-

(a) User-item matrix

	f_0	f_1	f_2	f_3
I_0	1	1	0	0
I_1	1	0	1	0
I_2	1	0	0	0
I_3	1	0	0	1

(b) Item-feature matrix

Figure 1: Examples of matrices used in RSs.

Software development environments are starting to integrate RSs to assist developers in various software engineering activities, from reusing code to effective bug reports [23]. Examples of recommended items in these systems are method calls that can be useful in a certain context [33], software components that may be reused in a given situation [17], and required software artefacts [16].

Likewise, RSs for modelling notations have started to appear, such as an RS capable of semi-automatically creating the draft of web application specifications by reusing models of previous software cases [20], and an RS for model-driven software engineering which considers the history of past model changes [15].

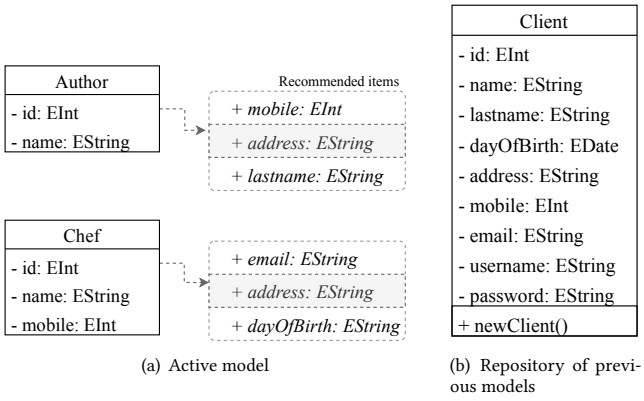
3 MOTIVATION AND OVERVIEW

In this paper, our goal is to devise mechanisms to simplify the development of RSs for LCDPs. Hence, in the following, Section 3.1 first presents a motivating running example, and then, Section 3.2 provides an overview of our proposal.

3.1 Motivating example

Many LCDPs, such as ZappDev[1], MetaDev[2] and Mendix[3], require identifying the concepts that the application being developed will manage, together with their attributes and methods. This information is typically provided by means of forms or using a diagrammatic notation, such as class diagrams.

Figure 2(a) shows an example where a user has created an Author class with a couple of attributes, and now wonders whether the class misses any important attributes. In other scenarios, the user may proactively ask other more experienced people or search on the Internet. However, this may be too demanding for the average LCDP user (i.e., the citizen developer). Therefore, in this case, we prefer to extend the LCDP with an RS that analyses previously developed similar classes to provide a ranked list of recommended attributes for the new class.



(a) Active model (b) Repository of previous models

Figure 2: Motivating example.

As an example, Figure 2(b) shows a Client class created some time before by a different user, which is conveniently stored in a repository together with many other classes. The RS would detect that the Client class is similar to the Author class because it defines the same attributes as this latter class (id, name). Hence, it would recommend adding Client's attributes (e.g., mobile and address) to Author. Moreover, the RS will use some algorithm to identify the attributes of Client that are the most relevant for Author. For instance, classes with attributes id and name may have been previously used more commonly with attributes like mobile than with attributes like password.

Altogether, in this example, the recommended items are attributes and methods, and the similarity between classes is based on the similarity of their attributes and methods (using criteria like name, visibility and type). However, other LCDPs may use alternative modelling notations, and the recommendation task may be different as well. Motivated by this situation, we envision a generic framework to facilitate the creation of RSs for specific LCDPs. The next subsection provides a high-level overview of its architecture.

[1]http://www.zappdev.com/
[2]https://metadev.pro/
[3]https://www.mendix.com/

3.2 Overview of the approach

Figure 3 shows the architecture of our proposed approach, which applies MDE techniques to the development of RSs. First (label 1), the recommender system designer provides the meta-model of the notation that will be the subject of the recommendation. In our running example, this is the meta-model of class diagrams. We assume the existence of a repository of models conformant to the meta-model, which will be used for the recommendation (label 2). Then, the designer uses a textual DSL (label 3) to define the meta-model elements that will play the roles of user, item and item features, as in traditional RSs. The DSL also permits customising other aspects of the RS, such as the maximum number of recommended items, the applied recommendation method, and the recommendation format that is the best fit for the task at hand.

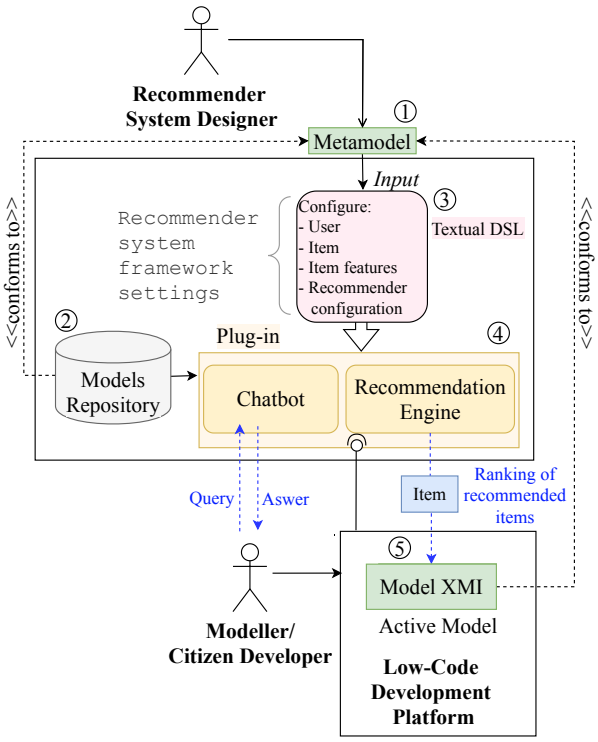


Figure 3: Overview of the proposed approach.

Starting from this information, our framework will generate a tailored RS available as a plug-in for the LCDP (label 4). This way, the citizen developers will be offered the recommendations within the LCDP environment (label 5). We foresee the provision of alternative ways to render the recommendations, such as tips over the diagram elements, example fragments, or by means of query-answer chatbots addressed in natural language.

4 PROPOSED APPROACH

This section details the main components of our proposal. Figure 4 shows the steps involved in the configuration and generation of an RS with our approach. In a first step, the RS designer needs to provide some data, specifically, the meta-model of the notation for

which the RS is to be developed, and the set of instance models to be used for training the RS. In step 2, the RS designer uses the DSL to configure the desired features of the RS. From this information, the RS designer can trigger the generation of the RS. This generation comprises steps 3 to 7, which are completely automated.

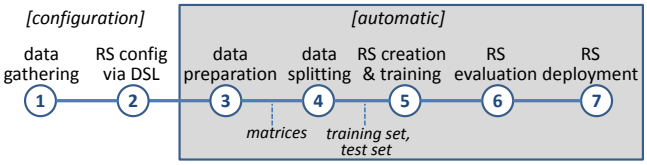


Figure 4: Overview of the process

In step 3, the data provided in step 1 are prepared to produce the user-item and item-feature matrices, considering the specific items and features indicated in the RS configuration. Then, the data are split into two sets (step 4): one is used for training the RS (step 5), and the other one is used for evaluating the accuracy of the RS after its training (step 6). Finally, in step 7, the resulting RS can be deployed and used to obtain lists of recommended items.

In the following subsections, we provide additional details of the DSL, the data preparation step and the recommendation engine.

4.1 Domain-specific language

This section describes the data gathering and the configuration of the RS via a DSL (steps 1 and 2 of Figure 4). We have designed a DSL to configure RSs for arbitrary languages that are defined by a meta-model. The DSL allows configuring the recommendation method, the data splitting method, the evaluation method, and the kind of elements to be recommended. The DSL provides a high-level syntax for this task, which avoids the RS designer the use of lower-level general-purpose programming languages like C or Java (typically more technical and complex) or the need to have deep expertise in libraries for RSs.

Figure 5 shows a meta-model that captures the main elements of the DSL. RecommenderConfiguration is the root class that contains the other classes, and specifies the name of the recommender, the meta-model of the notation for which the RS is being defined, and a set of instance models conformant to this meta-model. The instance models will be used to train the recommender.

The RecommendationMethod class permits selecting the recommendation methods of interest (e.g., item popularity, collaborative filtering, content-based) and configuring their parameters (e.g., the neighbourhood size for collaborative filtering methods). The SplitMethod class allows customising how to split the set of provided instance models for training and testing the RS. In particular, it defines the split type (e.g., cross-validation, random), the number of folds (if needed), the splitting method (per-user or per-item), and the percentage of data used for training the RS (the rest of the data will be automatically assigned for testing). The EvaluationMethod class defines all the configuration related to the evaluation of the RS, namely, the metrics used to evaluate the RS (e.g., precision, recall, F1), the maximum number of recommended items, and the relevance threshold. The EvaluationResult class represents the values of the evaluation metrics after executing each selected

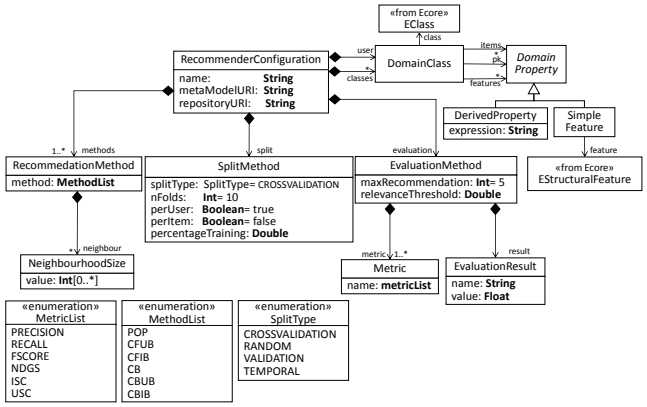


Figure 5: Meta-model of the DSL.

recommendation method. DomainClass allows specifying the type of the model elements that will play the role of user in the context of the RS. Likewise, DomainProperty is used to specify the type of the items to be recommended, which can be either features (attributes or references) of the specified DomainClass or derived features via expressions.

Listing 1 illustrates the textual concrete syntax that we have devised for the DSL, and which is currently being developed. The listing configures the RS for our running example. For clarity, we assume our RS is to be developed for simple class diagrams, conformant to the simple meta-model shown in Figure 6. This meta-model allows the specification of ClassDeclarations, AttributeDeclarations and MethodDeclarations.

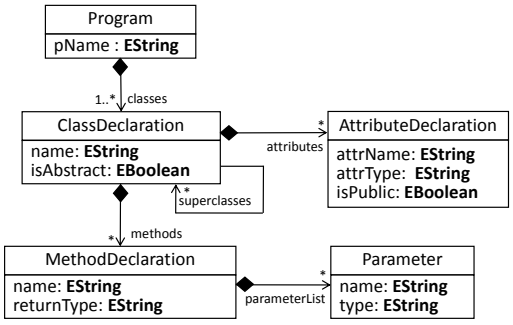


Figure 6: Simple meta-model for class diagrams.

In Listing 1, lines 1–2 identify the *meta-model* of the language the RS is built for (cf. Figure 6), and the URL of a *repository* of instances of this meta-model (step 1 in Figure 4). The following lines configure the RS (step 2 in Figure 4). Lines 5–6 specify the meta-model elements that will play the roles of *users* and *items* in the RS. These elements must belong to the meta-model provided in line 1. The listing sets the class ClassDeclaration as the **User** of the RS, while its attributes, methods and superclasses are set as the **Items** of ClassDeclaration. This means that the RS will be able to recommend these three kinds of items for a given class.

Then, lines 9–18 define the primary key used to identify each user and item in the RS, as well as the features used for comparing

Listing 1: Example of recommender system configuration

```
1  Metamodel: "/SimpleOOPL.ecore"
2  Repository: "/Instances/"
3
4  //Definition of user and items
5  Users: ClassDeclaration {
6    Items: attributes, methods, superclasses; }
7
8  //Definition of primary keys (pks) and features
9  ClassDeclaration {
10   pk: name; }
11
12 AttributeDeclaration {
13   pk: attrName;
14   features: attrName, attrType; }
15
16 MethodDeclaration {
17   pk: name;
18   features: name, returnType; }
19
20 //Recommender preferences
21 Recommendations {
22   //split configuration
23   Split {
24     splitType: CrossValidation;
25     nFolds: 10;
26     perUser: true;
27     percentageTraining: 0.8; }
28
29   //methods configuration
30   Methods {
31     collaborativeFiltering: pop, cfub(2,3,5,10), cfib;
32     contentBased: cb;
33     hybrid: cbub(2,3,5,10), cbib(2,3,5,10); }
34
35   //evaluation configuration
36   Evaluation {
37     metrics: precision, recall, f1, ndgs, isc, usc;
38     maxRecommendations: 5;
39     relevanceThreholds: 0.5; }}
```

users or items of the same type. For instance, lines 12–14 specify this information for the item AttributeDeclaration. In particular, its attribute attrName will be used as its primary key, and the features attrName and attrType will be used for the comparison of attribute declarations.

The remainder of the listing declares recommender preferences. The **Split** fragment (lines 23–27) configures the application of the *cross-validation* split method type with 10 folds, following a *per user* technique, and using 80% of the input data as training data. The **Methods** fragment (lines 30–33) selects the recommendation methods to apply and evaluate. Among others, the DSL designer has selected some collaborative filtering methods such as *pop* (item popularity) and *cfub* (collaborative filtering user base with 2, 3, 5 and 10 neighbours). Section 5.2.2 will describe these methods. Finally, the **Evaluation** fragment (lines 36–39) selects the evaluation protocol. In particular, line 37 chooses the metrics to be used for the evaluation, line 38 specifies the number of items to recommend, and line 39 defines a relevance threshold.

4.2 Data preparation

This section describes the data preparation for the RS (step 3 in Figure 4). Once the RS has been configured using the DSL, the first step that our framework performs is preparing the data for building and evaluating the RS. Figure 7 shows the methodology for this. First, the framework retrieves the collection of models specified with the DSL (1). Then, it extracts the model objects corresponding to the configured types of users, items and item features (2). Finally, it generates a user-item matrix and an item-feature matrix for them (3). As we explained in Section 2, the user-item (resp. item-feature) matrix contains the users (resp. items) as rows and the items (resp. features) as columns. Then, each cell is set to 1 if the user (resp. item) has the item (resp. feature), and to 0 otherwise.

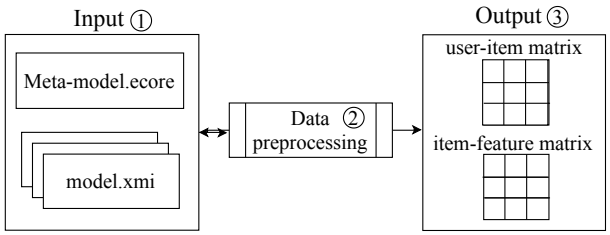
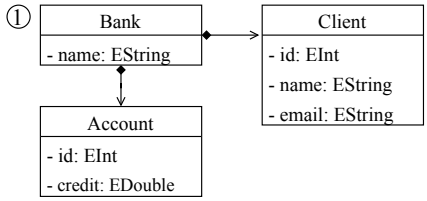


Figure 7: Data preparation steps.

Figure 8 shows an example of data preparation for the running example. To ease understanding, we assume that there is a single class diagram with three classes (1). The table with label 2 shows the extracted users, i.e., the three classes. The table with label 3 shows the extracted items, i.e., each different attribute and method declaration. The item comparison is based on the features selected in Listing 1 (e.g., attrName and attrType for attributes). The table with label 4 shows the value of those item features. From this information, the framework builds the user-item matrix shown to the right (5), where each row represents a class, and each column represents an attribute, method or superclass. The figure also shows the generated item-feature matrix (6).



②	user0	user1	user2
name (**pk**)	Bank	Account	Client

③	item0	item1	item2	item3
attrName (**pk**)	id	credit	name	email

④	feature0	feature1	feature2	feature3
attrName	id	credit	name	email
attrType	EInt	EDouble	EString	EString

	i0	i1	i2	i3
u0	0	0	1	0
u1	1	1	0	0
u2	1	0	1	1

User-item matrix ⑤

	f0	f1	f2	f3
i0	1	0	0	0
i1	0	1	0	0
i2	0	0	1	0
i3	0	0	0	1

Item-feature matrix ⑥

Figure 8: Example of data preparation.

4.3 Recommendation engine

This section describes the data splitting, the RS creation and training, the RS evaluation, and the RS deployment (steps 4–7 in Figure 4).

The matrices generated by the data preparation step are used to build the RS, as presented in Figure 9. Specifically, our framework splits the provided data into two sets: one for training the RS, and

the other to evaluate the quality of the resulting system (2) (step 4 in Figure 4). The splitting is made according to the specified protocol (see lines 23–27 in Listing 1). Next, the framework uses the configured recommendation methods (3) to train the RS with the training set (4) (step 5 in Figure 4). The RS designer may have configured several methods, as in lines 30–33 in Listing 1, and hence, several candidate RSs may be generated. Then, the test set is applied to each candidate system, and a score is computed based on the obtained results in each case (5). Finally, each candidate RS is evaluated (step 6 in Figure 4) according to the specified metrics (6, see lines 36–39 in Listing 1), and the results are made available for the designer inspection.

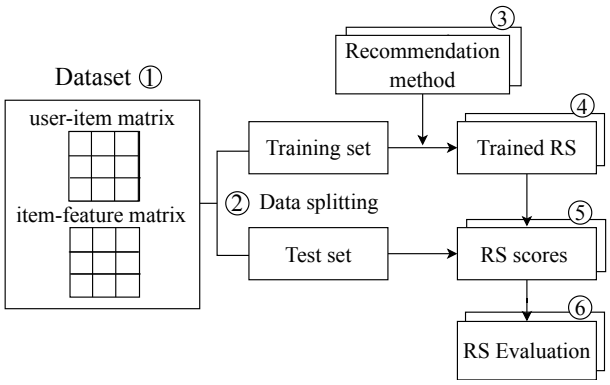


Figure 9: Steps to build the recommendation engine.

In the long term, we envision an intelligent framework that is able to suggest the best configuration for the target recommendation task and the available data (step 7 in Figure 4). This would free the RS designer from having to possess deep expertise in RS techniques.

5 PROOF OF CONCEPT

In this section, we present some initial results of our envisioned framework, which is being developed using Java and the Eclipse Modeling Framework (EMF) [29]. Effectively, the latter means that our RSs are applicable to languages defined by an Ecore meta-model. We currently have automated support for data preparation, data splitting, RS training and RS evaluation. The configuration data must be provided programmatically though, as the configuration DSL, while designed, is still under development. The deployment of the generated RS in an LCDP is also future work.

To have an initial assessment of our framework, we have applied it to the construction of the RS used throughout the paper as a running example. The RS would be integrated into an LCDP and would suggest attributes, methods and superclasses that may be added to new classes, based on the definition of other similar classes. By means of this experiment, we aim to answer the research questions stated in the introduction.

Next, Section 5.1 details the datasets used in the evaluation, Section 5.2 reports on the experiment, Section 5.3 presents the results and the answer to the research questions, and Section 5.4 concludes by identifying some threats to validity.

5.1 Experiment setup

We run the experiment on three datasets. Table 1 shows some size metrics of them (number of models, users, items and item features).

Table 1: Description of the datasets.

	Synthetic	SyntheticExtended	AtlanEcore
Num. models	29	58	300
Num. users	150	181	6555
Num. items	412	520	4338
Num. features	438	557	4867

The *Synthetic* dataset contains 29 models conformant to the running example meta-model (cf. Figure 6). The models were created manually using EMF[4]. These models are based on class diagram examples from the internet. We have made sure that the models created have all the characteristics normally present on class diagrams, such as attributes, methods and inheritance hierarchies.

The *SyntheticExtended* dataset extends the first one with further models which are similar to those in the *Synthetic* dataset but substituting the name of some model elements by synonyms.

Finally, since meta-models are similar to class diagrams, our third dataset (*AtlanEcore*) is composed of 300 Ecore meta-models from the AtlanEcore Zoo[5]. This is an open-source repository of Ecore meta-models, which are conformant to the Ecore.ecore meta-meta-model. With this last dataset, we want to validate the versatility of our proposal.

The configuration of the RS for the first two datasets was the one shown in Listing 1. The configuration for the *AtlanEcore* dataset was similar but using types from Ecore.ecore (i.e., setting `EClass` as the user of the RS; `eAttributes`, `eOperations` and `eSuperTypes` as the items; and so on).

5.2 Experiment

Next, we describe the splitting protocol applied to the data of our experiment (Section 5.2.1), the evaluated recommendation methods (Section 5.2.2), and the used evaluation metrics (Section 5.2.3).

5.2.1 Data splitting. We used *10-fold cross-validation* with 80% of the data as a training set, and the remainder 20% as a test set (cf. lines 23–27 in Listing 1). We followed a *per-user* method, whereby the training and test sets are built per available user (i.e., for each class, it takes 80% of its items for training and the rest for testing). Using 10-fold cross-validation avoids over-specialization. This is so as the training set is split into 10 subsets, the training is performed 10 times taking one of the subsets for testing and the others for training, and finally, the average performance of the 10 learned RSs is reported.

5.2.2 Recommendation methods. We trained the RS using a variety of collaborative, content-based, and hybrid recommendation methods (cf. lines 30–33 in Listing 1). For reproducibility, we used the RankSys framework[6] to implement the methods. The recommendation consisted of the top 5 highest-rated items.

[4] https://www.eclipse.org/modeling/emf/
[5] https://web.imt-atlantique.fr/x-info/atlanmod/index.php?title=Ecore
[6] http://ranksys.org/

In particular, the collaborative methods will recommend to users (i.e., to classes) items (i.e., attributes, methods, superclasses) that were rated (i.e., used) by like-minded users. The similarity between users and items is based on rating patterns [9]. In the experiment, we evaluated both user-based and item-based *k-nearest neighbour* (*k*-NN) heuristics. The item-based approach (**cfib**) creates neighbourhoods by exploiting the rating-based similarities between items, and the user-based approach (**cfub**) computes similarities between users [9]. We have used the cosine as a similarity metric. We tested neighbourhoods of size k = 2, 3, 5, 10. To refer to a specific instance of a method, we concatenate the value of k to its name. For instance, **cfub3** refers to collaborative user-based k-NN size k = 3. As a baseline, we also evaluated the item popularity method (**pop**).

The content-based method (**cb**) will recommend to users (i.e., to classes) items (i.e., attributes, methods, superclasses) similar to the ones liked (i.e., used) by the user. In this case, the similarity is computed based on profiles built from textual information [9]. The item features correspond to text features extracted from the items, and the recommendations are based on similarities in the text feature space. In the experiment, we used the name and data type as features of attribute declarations; the name and return type as features of method declarations; and the name of superclasses.

Finally, the hybrid methods exploit both rating and text features by combining content-based and collaborative methods [9]. We considered the methods **cbub** and **cbib**, which combine either user-based (**cfub**) or item-based (**cfib**) collaborative filtering with content-based similarity (**cb**). We tested neighbourhoods of size k = 2, 3, 5, 10. As before, we concatenate the name of the method and the value of k. For instance, **cbub5** refers to content-based user-based k-NN size k = 5.

5.2.3 Evaluation metrics. We analysed the performance of the resulting RSs using some ranking-based, coverage and diversity metrics typically used in RSs [22] (cf. lines 36–39 of Listing 1). The metrics were implemented in the RiVaL framework[7].

As mentioned in Section 2, the selection of metrics depends on the faced recommendation problem. In particular, since our RS should provide an ordered list of recommended items, we used the classical ranking metrics *precision*, *recall* and *F1*. *Precision* is the percentage of the recommended items that are relevant; *recall* is the percentage of relevant items included in the recommendation list; *F1* is a harmonic mean of precision and recall.

To measure coverage, we used the metrics *USC* (user space coverage) and *ISC* (item space coverage). *USC* measures the percentage of users that the RS can recommend, and *ISC* the diversity in terms of the popularity of what is recommended.

Finally, to measure the quality of the recommended list, which should contain just the most relevant items, we used the metric *nDCG* (normalized discounted cumulative gain). This metric penalises when the most relevant items are not at the top of the list.

5.3 Experiment results and research questions

Table 2 shows the results of the experiment. The rows contain the recommendation methods used to train the RS, and the columns show their performance metrics. We can see that the metric values are drastically different depending on the dataset. The *AtlanEcore* dataset has the best overall performance. A possible reason is that this is the largest dataset among the three. Next, we answer our initial research questions.

5.3.1 ***RQ1*** *Can a recommender system help in class modelling tasks?* In order to answer this question, we analyse the performance of the recommendation methods. Specifically, we look at their *precision*, *recall* and *F1* values in Table 2, as they give a measure of the ranking quality. With regards to our evaluation methodology, where the task is recommending the most relevant items in the test set, the obtained performance is relevant according to the literature [14, 19, 22, 28]. In the *AtlanEcore* dataset, the highest *F1* value was 0.289 for the **cfub2** method. This shows that we can build an RS that helps in class modelling by recommending valuable attributes, methods and superclasses for a given class.

5.3.2 ***RQ2*** *Which recommendation method of relevant attributes, methods and superclasses has the best performance?* In the *SyntheticExtended* and *AtlanEcore* datasets, the collaborative filtering methods obtained the best performance. In particular, **cfub2** has the best results, as the *F1* measure is 0.135 and 0.289 for the *SyntheticExtended* and *AtlanEcore* datasets, respectively. Conversely, among the collaborative methods, the baseline **pop** has the worst performance (e.g., 0.018 precision and 0.083 recall on the *AtlanEcore* dataset). When it comes to coverage, **pop** has a very high *USC* value (1.000), as it recommends the most popular items to all users. However, it posses a low *ISC* (0.002), which suggests a very low diversity.

As for the *Synthetic* dataset, the best result was obtained for the hybrid method **cbib2**, followed by all other collaborative methods.

5.3.3 ***RQ3*** *Can hybrid approaches be beneficial for the recommendation of attributes, methods and superclasses?* When analysing the hybrid methods applied in this experiment, we observe that some performed very well. For instance, in the *Synthetic* dataset, **cbib2** performed even better than the collaborative methods, obtaining 0.095 *precision*, 0.199 *recall*, and an *F1* value of 0.129.

5.3.4 ***RQ4*** *Which method performs better when considering user and item coverage in the recommendation of relevant attributes, methods and superclasses?* We observe a compromise between the coverage metrics *USC* and *ISC*, and the ranking-based metrics. The methods with low precision and recall, like most of the hybrid ones, report high user coverage and low item coverage. A good user coverage comes at the cost of losing item diversity when compared to collaborative methods.

5.4 Threats to validity

In the following, we discuss internal and external threats that may affect the validity of the findings of our experiment. Internal validity is the extent to which there is a causal relationship between our study and the extracted conclusions [21]. In this respect, some of our datasets have a low number of models, which may lead to cases where there are no similar classes to a particular one. In addition, the in-house datasets were created by hand, and hence, there is the risk that we have inadvertedly introduced some bias. To address this threat, the experiment included the *AtlanEcore* dataset,

[7] http://rival.recommenders.net/

Table 2: Results of the experiment. The best values are shown in bold.

	Synthetic						SyntheticExtended						AtlanEcore					
Method	prec.	recall	F1	nDCG	ISC	USC	prec.	recall	F1	nDCG	ISC	USC	prec.	recall	F1	nDCG	ISC	USC
pop	0.048	**0.221**	0.079	**0.177**	0.015	**1.000**	0.046	0.207	0.076	0.170	0.015	**1.000**	0.018	0.083	0.029	0.055	0.002	**1.000**
cfub2	0.060	0.185	0.091	0.144	0.035	0.719	**0.093**	0.242	**0.135**	0.179	0.043	0.698	**0.241**	0.362	**0.289**	**0.323**	0.048	0.332
cfub3	0.054	0.190	0.084	0.145	**0.036**	0.802	0.088	0.256	**0.132**	0.188	0.046	0.802	**0.211**	**0.367**	**0.268**	**0.322**	0.055	0.372
cfub5	0.054	0.206	0.085	**0.165**	**0.036**	0.898	0.083	**0.276**	0.128	**0.202**	0.048	0.837	0.179	**0.368**	0.241	0.321	0.061	0.415
cfub10	**0.066**	0.193	**0.099**	0.146	0.032	0.600	0.074	**0.289**	0.118	**0.219**	**0.049**	0.919	0.140	0.347	0.200	0.297	**0.067**	0.482
cfib	0.053	**0.207**	0.085	0.147	**0.038**	0.901	0.064	0.239	0.101	0.172	**0.049**	0.921	0.092	0.273	0.138	0.225	0.063	0.627
cb	0.018	0.086	0.030	0.086	0.008	**1.000**	0.018	0.086	0.030	0.085	0.006	**1.000**	0.005	0.022	0.008	0.010	0.001	**1.000**
cbub2	0.016	0.015	0.016	0.016	0.002	0.968	**0.096**	0.176	0.125	0.124	0.033	0.633	0.200	0.246	0.221	0.220	0.035	0.311
cbub3	0.016	0.032	0.022	0.026	0.005	0.968	0.065	0.196	0.098	0.142	0.040	0.745	0.155	0.259	0.194	0.230	0.048	0.410
cbub5	0.016	0.057	0.025	0.039	0.010	0.968	0.057	0.203	0.089	0.147	0.043	0.896	0.113	0.276	0.160	0.238	0.058	0.483
cbub10	0.016	0.070	0.026	0.044	0.012	0.968	0.052	0.211	0.083	0.158	0.043	0.993	0.079	0.269	0.122	0.228	**0.065**	0.558
cbib2	**0.095**	0.199	**0.129**	0.131	0.026	0.539	0.014	0.010	0.012	0.011	0.002	0.973	0.001	0.001	0.001	0.001	0.000	0.697
cbib3	0.049	0.166	0.075	0.120	0.030	0.634	0.013	0.020	0.016	0.018	0.004	0.973	0.001	0.002	0.002	0.002	0.000	0.697
cbib5	0.036	0.131	0.056	0.098	0.033	0.864	0.013	0.039	0.019	0.027	0.008	0.973	0.001	0.005	0.002	0.003	0.001	0.697
cbib10	0.032	0.138	0.051	0.112	0.034	**1.000**	0.013	0.049	0.020	0.031	0.010	0.973	0.001	0.006	0.002	0.004	0.001	0.697

developed by a third-party. External validity refers to the extent to which conclusions of an experiment can be generalized [21]. Our experiment considers a very specific task and language, which is the recommendation of attributes, methods and superclasses for class diagrams and meta-models. In the future, we plan to implement and evaluate our proposal with other recommendation tasks and languages. Additionally, we will analyse the performance of other datasets.

6 RELATED WORK

In this section, we place our work regarding the state-of-the-art. First, in Section 6.1, we describe related works on RSs and model assistant approaches for MDE. Afterwards, in Section 6.2, we explore MDE frameworks to customise RSs for a particular context, in order to reduce time and effort in a constantly changing world of recommendation techniques and approaches.

6.1 Recommender systems for MDE

Some approaches aim at providing semantically related terms and context-sensitive information for a modelling task. Rickauer et al. [2, 3] developed DOMORE, an RS for domain modelling based on an extensive knowledge base of domain-specific terms and their relationships. The recommendations are built based on a term ocurrence technique. Similarly, Mora et al. [18, 26, 27] implemented EXTREMO, a tool supporting the uniform query of heterogeneous sources. This tool facilitates the reuse of information in the form of semantic-related terms. While these approaches suggest valuable terms for the modelling task, suggestions are only based on the semantic relations between the terms used in the given context. Other techniques that could enrich these solutions, such as exploiting the similarity between models, are not employed. Additionally, these tools target a specific modelling task, while our envisioned framework aims to be generic and configurable for arbitrary modelling recommendation tasks, to be embedded in LCDPs.

Other works focus on recommendations for UML diagrams. For example, Cerqueira et al. [10] proposed a content-based approach for recommending behavioural features for UML sequence diagrams. They compared their approach with a bag of words model and found no statistical difference between them. Also, Paydar et al. [20] developed a prototype capable of semi-automatically create a draft web application from a list of functional requirements. For this purpose, their approach is based on a repository containing semantic representations of models of previously developed web applications. Then, a semantic similarity algorithm was employed on a hybrid approach. While these works tackled important modelling tasks and were proved to be useful, they serve a specific modelling language. Instead, the targeted language is customisable in our framework.

Some researchers have proposed language-independent recommendation solutions for modelling tasks. Stephan [30] proposed SimVMA, which helps modellers find models or operations considered relevant to them. The approach is based on detecting clones between the in-progress model and similar models, using the Simone clone detector for detecting type 3 near-miss clones. While the approach was exemplified for Simulink, the envisioned approach is not Simulink-specific. Also, Kögel [15] proposes an approach to analyse the history of past model changes to suggest recommendations. It is based on analysing how other users changed the models over their lifetime. The recommendations are implemented as Henshin rules. The authors leave as future work the possibility of applying machine learning, heuristic search algorithms, association rules and decision trees. Even though these works are planning on frameworks for different modelling languages, the recommendation technique is fixed, and the recommendations cannot be customised according to the needs of the RS designer, as we aim to do with our DSL.

6.2 MDE for recommender system generation

In [25], Rojas et al. proposed a model-driven framework to develop mobile RSs of geographic points of interest. The framework helps developers to specify the structural, behavioural and navigational

aspects of the RS. It permits customising the user preferences, similarity metrics and similarity formula. It incorporates a collaborative filtering algorithm and their combination with content-based and location-based algorithms. The generated RS suggests the 10 most relevant points of interest.

In [24], the authors applied a similar solution to recommend trips and tours. In this case, the framework applied an item-to-item similarity method and allowed setting the user preferences, the similarity metrics, and the selection and order of the recommendations. Additionally, some MDE approaches have emerged to support non-expert users on applying data mining. For example, Espinosa at al. [11, 12] reuse the past experiences of data mining experts with the application of classification techniques and data. An RS computes the accuracy for a given new dataset and recommends the one with the best performance. The customisable parameters in this framework are the data mining task to perform, the metric used to evaluate the recommender performance, the validation testing method, and the mining algorithm to execute. Even though these solutions offer the flexibility and benefits of MDE, they generate recommenders either for e-commerce or data mining applications.

7 CONCLUSION AND FUTURE WORK

In this paper, we have introduced a generic framework to automate the construction of RSs that assist "citizen developers" in creating software applications via LCDPs. The framework provides a DSL to customise the different aspects of the RS. These include the language the RS is built for, the recommendation algorithm and its parameters, and the evaluation method. We have illustrated the approach by creating a recommender of attributes, methods and superclasses, which we have evaluated on three datasets using different recommendation methods. Some of these methods performed quite well, with results similar to those reported in the literature. Overall, the main advantage of our proposal is the flexibility to define RSs according to the RS designer needs.

We are currently developing the concrete syntax of our DSL to configure the RSs. In the future, we plan to apply our framework to more languages and modelling tasks. Moreover, we plan to provide support for deploying the synthesised recommenders in the LCDPs. Specifically, we foresee the provision of a chatbot that integrated within the LCDP that citizen developers address to access the recommendations.

ACKNOWLEDGMENTS

This project has received funding from the EU Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 813884, the Spanish Ministry of Science (RTI2018-095255-B-I00) and the R&D programme of Madrid (P2018/TCS-4314).

REFERENCES

[1] Gediminas Adomavicius and Alexander Tuzhilin. 2005. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Trans. on Knowl. and Data Eng.* 17, 6 (2005), 734–749.
[2] Henning Agt-Rickauer, Ralf-Detlef Kutsche, and Harald Sack. 2018. DoMoRe - A recommender system for domain modeling. In *MODELSWARD*. SciTePress, 71–82.
[3] Henning Agt-Rickauer, Ralf-Detlef Kutsche, and Harald Sack. 2019. Automated recommendation of related model elements for domain models. In *Model-Driven Engineering and Software Development*. Springer International Publishing, 134–158.
[4] Bruno Antunes, Joel Cordeiro, and Paulo Gomes. 2012. An approach to context-based recommendation in software development. In *RecSys*. ACM, 171–178.
[5] B. Ashok, Joseph M. Joy, Hongkang Liang, Sriram K. Rajamani, Gopal Srinivasa, and Vipindeep Vangala. 2009. DebugAdvisor: A recommender system for debugging. In *ESEC/SIGSOFT FSE*. ACM, 373–382.
[6] Lorenzo Bettini. 2013. *Implementing Domain-Specific Languages with Xtext and Xtend* (2 ed.). Packt Publishing Ltd.
[7] Saad bin Abid, Vishal Mahajan, and Levi Lucio. 2019. Machine learning for learnability of MDD tools. In *SEKE*. 355–468.
[8] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-Driven Software Engineering in Practice* (2nd ed.). Morgan and Claypool Publishers.
[9] Iván Cantador, María E Cortés-Cediel, Miriam Fernández, and Harith Alani. 2018. What's going on in my city?: recommender systems and electronic participatory budgeting. *RecSys* (2018), 219–223.
[10] Thaciana G. O. Cerqueira, Franklin Ramalho, and Leandro Balby Marinho. 2016. A content-based approach for recommending UML sequence diagrams. In *SEKE*. 644–649.
[11] Roberto Espinosa, Diego García-Saiz, Marta E. Zorrilla, José Jacobo Zubcoff, and Jose-Norberto Mazón. 2013. Development of a knowledge base for enabling non-expert users to apply data mining algorithms, In SIMPDA. *CEUR Workshop Proceedings* 1027, 46–61.
[12] Roberto Espinosa, Diego García-Saiz, Marta E. Zorrilla, José Jacobo Zubcoff, and Jose-Norberto Mazón. 2019. S3Mining: A model-driven engineering approach for supporting novice data miners in selecting suitable classifiers. *Computer Standards and Interfaces* 65 (2019), 143–158.
[13] Asela Gunawardana and Guy Shani. 2015. Evaluating recommender systems. In *Recommender Systems Handbook*. Springer, 265–308.
[14] Bernd Heinrich, Marcus Hopf, Daniel Lohninger, Alexander Schiller, and Michael Szubartowicz. 2019. Data quality in recommender systems: The impact of completeness of item content data on prediction accuracy of recommender systems. *Electronic Markets* (2019), 1–21.
[15] Stefan Kögel. 2017. Recommender system for model driven software development. (2017), 1026–1029.
[16] Walid Maalej and Alexander Sahm. 2010. Assisting engineers in switching artifacts by using task semantic and interaction history. *RSSE@ICSE* (2010), 59–63.
[17] Frank McCarey, Mel Ó Cinnéide, and Nicholas Kushmerick. 2005. RASCAL: A recommender agent for agile reuse. *Artificial Intelligence Review* 24, 3-4 (2005), 253–276.
[18] Ángel Mora Segura and Juan de Lara. 2019. Extremo: An Eclipse plugin for modelling and meta-modelling assistance. *Sci. Comput. Program.* 180 (2019), 71–80.
[19] Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, Lina Ochoa, Thomas Degueule, and Massimiliano Di Penta. 2019. FOCUS: A recommender system for mining API function calls and usage patterns. In *ICSE*. IEEE, 1050–1060.
[20] Samad Paydar and Mohsen Kahani. 2015. A semantic web enabled approach to reuse functional requirements models in web engineering. *Autom. Softw. Eng.* 22, 2 (2015), 241–288.
[21] Samad Paydar and Mohsen Kahani. 2015. A semi-automated approach to adapt activity diagrams for new use cases. *Inf. Softw. Technol.* 57 (2015), 543–570.
[22] Francesco Ricci, Lior Rokach, and Bracha Shapira. 2015. *Recommender Systems Handbook* (2 ed.). Springer US.
[23] Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann. 2014. *Recommendation Systems in Software Engineering*. Springer-Verlag Berlin Heidelberg 2014.
[24] Gonzalo Rojas, Francisco Dominguez, and Stefano Salvatori. 2009. Recommender systems on the Web: A model-driven approach. In *E-Commerce and Web Technologies*, Tommaso Di Noia and Francesco Buccafurri (Eds.). Springer Berlin Heidelberg, 252–263.
[25] Gonzalo Rojas and Claudio Uribe. 2013. A conceptual framework to develop mobile recommender systems of points of interest. In *SCCC*. IEEE Computer Society, 16–20.
[26] Ángel Mora Segura, Juan de Lara, Patrick Neubauer, and Manuel Wimmer. 2018. Automated modelling assistance by integrating heterogeneous information sources. *Comput. Lang. Syst. Struct.* 53 (2018), 90–120.
[27] Ángel Mora Segura, Ana Pescador, Juan de Lara, and Manuel Wimmer. 2016. An extensible meta-modelling assistant. In *EDOC*. IEEE Computer Society, 1–10.
[28] Ritu Sharma, Dinesh Gopalani, and Yogesh Meena. 2017. Collaborative filtering-based recommender system: Approaches and research challenges. In *ICICT*. 1–6.
[29] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework, 2^{nd} Edition*. Addison-Wesley Professional. See also http://www.eclipse.org/modeling/emf/.
[30] Matthew Stephan. 2019. Towards a cognizant virtual software modeling assistant using model clones. In *ICSE (NIER)*. IEEE / ACM, 21–24.

[31] Panagiotis Symeonidis, Alexandros Nanopoulos, and Yannis Manolopoulos. 2008. Providing justifications in recommender systems. *IEEE Trans. Syst. Man Cybern. Part A* 38, 6 (2008), 1262–1272.

[32] Massimo Tisi, Jean-Marie Mottu, Dimitrios S. Kolovos, Juan de Lara, Esther Guerra, Davide Di Ruscio, Alfonso Pierantonio, and Manuel Wimmer. 2019. Low-comote: Training the next generation of experts in scalable low-code engineering platforms. In *STAF (Co-Located Events) (CEUR Workshop Proceedings, Vol. 2405)*. CEUR-WS.org, 73–78.

[33] Masateru Tsunoda, Takeshi Kakimoto, Naoki Ohsugi, Akito Monden, and Kenichi Matsumoto. 2005. Javawock: A Java class recommender system based on collaborative filtering. *SEKE*, 491–497.

[34] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart Kats, Eelco Visser, and Guido Wachsmuth. 2013. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages.* dslbook.org.