# Automating the Synthesis of Recommender Systems for Modelling Languages

Lissette Almonte
Universidad Autónoma de Madrid
Madrid, Spain

Sara Pérez-Soler
Universidad Autónoma de Madrid
Madrid, Spain

Esther Guerra
Universidad Autónoma de Madrid
Madrid, Spain

Iván Cantador
Universidad Autónoma de Madrid
Madrid, Spain

Juan de Lara
Universidad Autónoma de Madrid
Madrid, Spain

## Abstract

We are witnessing an increasing interest in building recommender systems (RSs) for all sorts of Software Engineering activities. Modelling is no exception to this trend, as modelling environments are being enriched with RSs that help building models by providing recommendations based on previous solutions to similar problems in the same domain. However, building a RS from scratch requires considerable effort and specialized knowledge.

To alleviate this problem, we propose an automated approach to the generation of RSs for modelling languages. Our approach is model-based, and we provide a domain-specific language called Droid to configure every aspect of the RS (like the type and features of the recommended items, the recommendation method, and the evaluation metrics). The RS so configured can be deployed as a service, and we offer out-of-the-box integration of this service with the EMF tree editor. To assess the usefulness of our proposal, we present a case study on the integration of a generated RS with a modelling chatbot, and report on an offline experiment measuring the precision and completeness of the recommendations.

***CCS Concepts:*** • **Software and its engineering → Software notations and tools**; **Designing software**; • **Information systems → Information retrieval**.

***Keywords:*** Modelling Languages, Model-Driven Engineering, Domain-Specific Languages, Recommender Systems

**ACM Reference Format:**
Lissette Almonte, Sara Pérez-Soler, Esther Guerra, Iván Cantador, and Juan de Lara. 2021. Automating the Synthesis of Recommender Systems for Modelling Languages. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering (SLE '21), October 17–18, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3486608.3486905


*SLE '21, October 17–18, 2021, Chicago, IL, USA*

ACM ISBN 978-1-4503-9111-5/21/10…$15.00
https://doi.org/10.1145/3486608.3486905

## 1 Introduction

Modelling plays a fundamental role in Software Engineering, especially in model-driven engineering (MDE) [9]. In this paradigm, models are actively used in the different development phases to specify, analyse, design, simulate and test the system to be built, among other activities.

Modelling is performed using modelling languages which can be general-purpose ones, like the Unified Modelling Language (UML) [56], or domain-specific languages (DSLs) tailored to a target domain [27, 58]. Today, sophisticated development environments and powerful language workbenches are the norm. However, modelling remains mostly a manual activity, which often does not profit from knowledge found in existing models, or the experience of engineers working on similar domains.

Recommender systems (RSs) [1] are information filtering systems that help users in choosing among a potentially large set of items (e.g., movies, songs or books). They aim at predicting the preferences of users to offer a prioritised list of potentially interesting items. They are widely used in all sorts of commercial and leisure applications, and their use in software engineering activities is increasing as well [46]. This way, we can find RSs that help in choosing appropriate third-party programming libraries [38, 55], recommend API method invocations [37], suggest code refactorings [13, 22], and assist on the evaluation of change impact analysis [8], to name a few. Recently, we are also witnessing an incipient interest to apply RSs to modelling (see, e.g., [2, 5, 10, 12, 15, 20, 23, 31, 34, 35]); however, their use in MDE is not the norm yet. One possible reason is that building RSs requires deep expertise in recommendation techniques, and involves an important development effort [4, 36].

With the aim to facilitate the adoption of RSs in MDE, we propose a model-driven solution to automate the synthesis of RSs for modelling languages. Based on the vision put forward in [3], our solution consists of a DSL called Droid supporting

the configuration of the kind of model elements to be recommended, and an engine that automates the evaluation of different recommendation methods against configurable metrics. The selected recommendation method is deployed as a service, which heterogeneous modelling clients can integrate. Currently, we provide an automated, out-of-the-box integration with the tree editor of the Eclipse Modeling Framework (EMF) [53], but additionally, the generated recommenders can be integrated with other modelling technologies. To assess this fact, we describe a case study of the integration of a recommender with a third-party modelling chatbot called Socio [42]. Finally, to assess the usefulness of our proposal, we report on an offline evaluation of a RS created with our approach over UML models. The experiment results are in line with RSs specifically created for class diagrams [10], but our approach does not require any programming.

*Paper organization.* Section 2 provides background on RSs. Section 3 overviews our approach, and Section 4 presents the Droid DSL. Section 5 details the technical architecture and tool support. Section 6 presents a case study incorporating a RS to a modelling chatbot, and an offline evaluation. Section 7 compares with related research, and Section 8 ends with the conclusions and future work.

## 2 Recommender Systems

Recommender systems have become a key component of a large and varied number of software applications. Nowadays, everyone is exposed to recommendation services on music (e.g., Spotify, Pandora) and video (e.g., Netflix, YouTube) streaming platforms, e-commerce sites (e.g., Amazon, eBay), and social networks (e.g., Facebook, Twitter), among others.

Common to all these applications, RSs analyse the activity of a typically very large group of users to provide them with personalised suggestions of options (items), based on the evidence observed about their interests and preferences. In this context, they provide advantages both for the users – whose experience is improved by receiving ideas about content to consume or products to buy – and the service providers – by promoting increased sales and customer loyalty, as customers are able to discover content or products that they would not have known otherwise.

In Software Engineering in general [21], and MDE in particular [4], RSs have also found a wide array of applications. Integrated in software design and development tools, recommendation services can assist on the creation [43], completion [2, 20], repair [6, 41, 50], search [35], and reuse [54] of artefacts, e.g., models, meta-models and transformations.

In these cases, the target *user* for whom recommendations are generated and her *preferences* (or *profile*) may have special meanings. For instance, it may refer to a *class* in a UML diagram for a recommender that is devoted to suggesting potential attributes and methods of interest for incomplete classes; the recommended *items* or artefacts are thus class attributes and methods, and the preferences and features that describe users and items can be the names and types of class attributes, methods and method arguments.

In general, the recommendations are generated based on content-based similarities between users and items [33], user-item preference (rating) patterns identified in the user community via collaborative filtering techniques [39], or both sources of information via hybridisation methods [11]. *Content-based* (CB) systems suggest items "similar" to those the target user preferred in the past, whereas *collaborative filtering* (CF) systems suggest items preferred by like-minded people. Moreover, CF approaches can consider either the similarities of the $k$ most similar users – known as nearest neighbours – to the target user (UBCF), or the similarities of the $k$ most similar items (IBCF) to the target user's items. Finally, typical hybrid approaches follow user- or item-based CF strategies exploiting content-based similarities (CBUB or CBIB) instead of rating-based similarities as CF does.

The quality of the recommendations can also be evaluated through different approaches [24]. User studies allow evaluating a recommender online, e.g., via A/B tests that capture the impact that recommendations have in real time. Offline experiments, by contrast, are conducted on datasets made of past user-item interactions, and are split into training and test data to build and evaluate a recommender, respectively. For both types of evaluations – online and offline – several metrics can be computed [7]. Typical metrics that measure the ranking quality of the recommendation lists are *precision*, i.e., the probability that a selected item is relevant; *recall*, i.e., the percentage of relevant items in the recommendation lists; *F1*, i.e., the harmonic mean of precision and recall; *MAP* (Mean Average Precision), i.e., the mean average precision over all the users; and *nDCG* (Normalized Discounted Cumulative Gain), which considers if the most useful items appear in the top positions of the recommendation lists. Other complementary metrics are *USC* (User Space Coverage), which measures the percentage of users that the RS can recommend, and *ISC* (Item Space Coverage), which measures the diversity of the recommendations.

As we will present in the subsequent sections, our model-based approach to automatically generate RSs allows configuring all the above-mentioned aspects: Recommendation methods, target user and item profiles, and offline evaluation methodologies and metrics (cf. Listing 1).

## 3 Overview of the Approach

To facilitate the construction of RSs for arbitrary modelling languages, we propose a model-based solution whose scheme is depicted in Fig. 1. Our solution permits customising a RS for a particular modelling language, assists in deciding which recommendation method works better for the recommendation task at hand, and generates a recommendation service that can be integrated with external modelling tools.

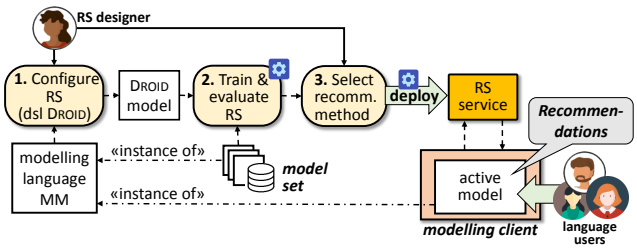



**Figure 1.** Overview of the approach.

Our approach makes available a DSL called Droid to configure the RS for the targeted modelling language. The approach assumes that the modelling language is defined by a meta-model. This way, in the first step, the designer of the RS uses Droid to select from the modelling language meta-model the elements that are to be recommended (e.g., attributes for a class, tasks for a process model). The DSL also permits specifying the candidate recommendation methods, the dataset used to train the recommenders, and the evaluation metrics used to rank the built recommenders. Section 4 will present Droid in detail.

Next, in the second step, our system automatically evaluates each selected recommendation method against the indicated metrics, using the provided dataset. The result is an interactive report with the value of each metric for the recommendation methods.

In the last step, the RS designer chooses a recommendation method, and the system automatically synthesizes a RS service that can be integrated within different modelling tools. Currently, our system provides full automatic support for deploying the RS within the Eclipse modelling tree editor [53], but other modelling clients are possible as well. Sections 5.4 and 6.2 will provide details on this out-of-the-box client integration, and others.

## 4 The Droid DSL

Droid is a textual DSL for the configuration of RSs for particular modelling languages. Fig. 2 shows its meta-model, which permits detailing the following aspects of a RS:

(i) The URI of the meta-model of the modelling language, and the repository containing the instance models to be used for training the RS. In Fig. 2, this is captured by the class RecommenderConfiguration and its attributes.

(ii) The class subject to recommendations, called *target* (class DomainClass and reference RecommenderConfiguration.target). The items to be recommended (class DomainProperty, its subclasses and reference DomainClass.items).

(iii) The way the objects of the target class and the items are identified in the models (references DomainClass.pk and DomainClass.features).

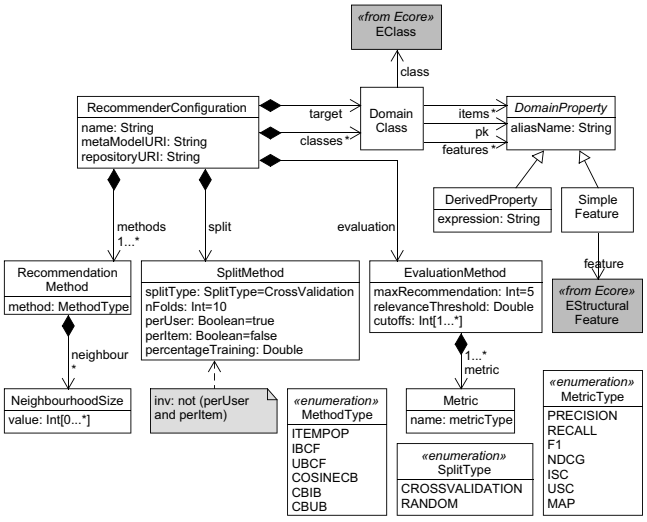



**Figure 2.** Meta-model of Droid.

(iv) The information about the candidate recommendation methods for the RS, and the metrics used to compare them. This is specified through the classes RecommendationMethod, SplitMethod and EvaluationMethod.

To illustrate Droid, we will use an example consisting of the configuration of a RS for UML class modelling. The RS will recommend attributes, operations and superclasses for a given class. Listing 1 shows its configuration using Droid.

In Listing 1, line 1 specifies the *name* of the recommender; line 2 specifies the *meta-model* that the RS will use (in this case, the meta-model of UML 2.0 class diagrams); and line 3 specifies the *repository* of models to be used to train and evaluate the selected candidate recommendation methods. The latter models need to conform to the specified meta-model. Since in our example, the training models are UML class models in the domain of libraries, the recommender is named "Literature Recommender."

Fig. 3 shows a small, simplified excerpt of the UML meta-model needed for our example: Classes, which contain both attributes (class Property) and Operations, and relate to their ancestors using the superClass derived reference.

The section "Target" in Listing 1 (lines 5–11) declares the *target* class of the recommendation and its relevant *items*. This way, the synthesised RS will provide recommendations for each declared item when invoked on objects of the target class. Line 6 specifies that the class Class is the recommendation target, and lines 7–9 specify three types of items to recommend: Attributes, methods and superclasses. Each item has a *name* (displayed to the user when the recommendation is performed), and the attributes or references leading from the target class to the items (in the example, ownedAttribute, ownedOperation and superClass, cf. Fig. 3). The meta-model of Droid also permits specifying derived properties via OCL

```
1  Recommender: "LiteratureRecommender"
2  Metamodel: "http://www.eclipse.org/uml2"
3  Repository: "/LiteratureRecommender/instances"
4
5  Target {
6    class Class {
7      item "attributes" : ownedAttribute;
8      item "methods" : ownedOperation;
9      item "super classes" : superClass;
10   }
11 }
12
13 Identifiers {
14   class Class {
15     pk feature name;
16   }
17   class Property {
18     pk feature name;
19     pk feature type;
20   }
21   class Operation {
22     pk feature name;
23     pk feature type;
24   }
25   class Type {
26     pk feature name;
27   }
28 }
29
30 Recommendations {
31   Methods {
32     collaborativeFiltering: ItemPop, IBCF(10,15,20,25,50,100),
33                             UBCF(10,15,20,25,50,100);
34     contentBased: CosineCB;
35     hybrid: CBIB(10,15,20,25,50,100), CBUB(10,15,20,25,50,100);
36   }
37   Split {
38     splitType: CrossValidation;
39     nFolds: 10;
40     perUser: true;
41   }
42   Evaluation {
43     metrics: Precision, Recall, F1, NDCG, ISC, USC, MAP;
44     cutoffs: 1,5,10,15,20;
45     maxRecommendations: 50;
46     relevanceThreshold: 0.5;
47   }
48 }
```

**Listing 1.** Defining a RS for UML class diagrams with Droid.

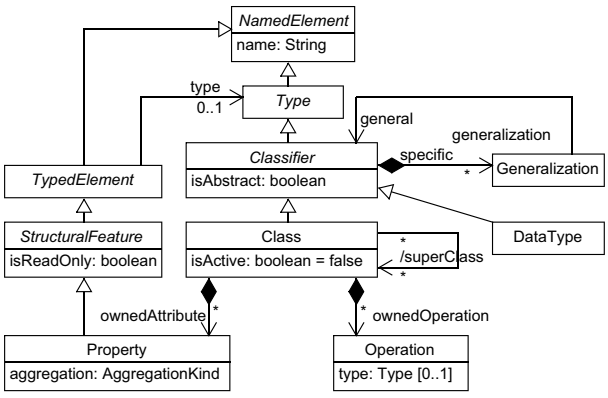



**Figure 3.** Simplified excerpt of the UML meta-model.

expressions [40] (class DerivedProperty in Fig. 2), though our current implementation does not support it yet.

The section "Identifiers" in lines 13–28 declares the identifiers and features of the involved classes. In our example, Classes are described by their name (line 15), which is their primary key (prefix pk), while Properties and Operations are identified by their name and type. The type of the latter is Type (cf. Fig. 3), and so, lines 25–27 declare its identifier.

The last section is "Recommendations", in lines 30–48. This enables the configuration of recommendation methods for the RS, and the way to evaluate them. This information is optional, since Droid provides default values in case the RS designer does not have the required expertise or is not interested in a fine-grained configuration of the evaluation process (see, e.g., the default values of class SplitMethod in Fig. 2). The section has three subsections: "Methods", "Split" and "Evaluation" (classes RecommendationMethod, SplitMethod and EvaluationMethod in Fig. 2).

First, the subsection "Methods" (lines 31–36) specifies the recommendation methods that the designer wants to experiment with to determine the best one for the case at hand. Lines 32–33 select the collaborative filtering methods *ItemPOP* (item popularity), *IBCF* (item-based collaborative filtering), and *UBCF* (user-based collaborative filtering). The latter two are configured with neighbourhood sizes of 10, 15, 20, 25, 50 and 100. Next, line 34 specifies the content-based method *CosineCB* (pure content-based method), and line 35 selects the hybrid methods *CBIB* (content-based item-based) and *CBUB* (content-based user-based) with neighbourhood sizes of 10, 15, 20, 25, 50 and 100. Overall, these are the six recommendation methods currently supported by Droid (cf. enumeration MethodType in Fig. 2).

The subsection "Split" (lines 37–41) specifies how to divide the dataset for the evaluation of the recommendation methods. The listing defines a 10-fold *cross-validation* split type, following a *perUser* technique. The split type refers to the approach to divide the data into training and test sets. *Cross-validation* divides the data into $k$ subsets, one used for test and the rest for training, and repeats the process assigning in each iteration the role of test set to each one of the $k$ subsets. Droid also supports *random* split, in which case, the percentage of data used for training/test must be given, and the sampling for the test/training sets is done randomly with a uniform distribution [45]. Splits can be built using either a *perUser* or a *perItem* technique. In the former case, the subsets are built per available user, while with *perItem*, they are built by available item. For example, a *perUser random* split type with 80% training percentage implies that 80% of the preferences of each user (i.e., 80% of the attributes, methods and superclasses of each class) will be used as the training set, and the remainder 20% as the test set.

The last part of the listing (subsection "Evaluation" in lines 42–47) describes the evaluation protocol. This includes the desired metrics for the evaluation (Precision, Recall, F1, nDCG, ISC, USC and MAP, cf. Section 2); the number of items in the top of the ranking that will be used to calculate the

metrics (cutoffs, line 44); the maximum number of items that the RS will recommend (maxRecommendations, line 45); and a threshold value for the rating of items used to determine whether an item is relevant and a good recommendation, or not (relevanceThreshold, line 46). This threshold defines a binary classification for the probability of a prediction to be true. In the listing, the relevance threshold of 0.5 implies that the rating values below 0.5 are considered false (irrelevant), and those equal to or above 0.5 are true (relevant).

## 5 Architecture and Tool Support

In this section, we introduce the architecture of Droid (Section 5.1), the tool support for RS design (Section 5.2), the generated recommendation service (Section 5.3), and the automatic integration with the EMF tree editors (Section 5.4).

### 5.1 Architecture

Fig. 4 shows the architecture of the Droid ecosystem, which comprises three parts. The first one is the *Droid Configurator*, which permits the configuration, evaluation and synthesis of RSs. The configurator provides an Eclipse textual editor for the DSL presented in Section 4, where the RS designer can configure the RS for a particular modelling language (label 1). The specified configuration is the input to the *RS Evaluator* (label 2), which relies on the external libraries *RankSys* [57] and *RiVal* [49] to evaluate the recommendation methods selected by the RS designer using Droid. RankSys is a framework for the implementation of recommendation algorithms, and RiVal is a toolkit for data splitting and evaluation of RSs. The results of each metric chosen by the RS designer are displayed in an Eclipse view (label 3). Section 5.2 will provide more details on the *Droid Configurator*.

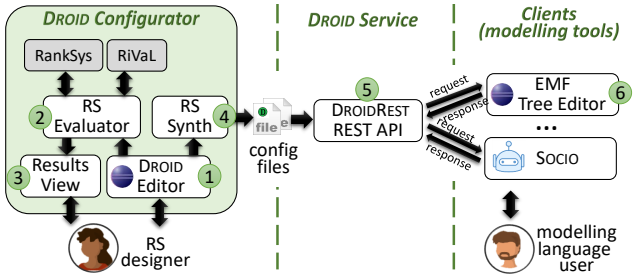



**Figure 4.** Architecture of Droid.

Based on the obtained results, the RS designer can select the preferred recommendation method, and a *RS Synthesizer* generates a set of configuration files out of the selection and the RS configuration (label 4). The configuration files are used by the second part of our ecosystem, which is the *Droid Service* (label 5). This is a generic recommendation REST API that can be customised for particular modelling languages using the configuration files generated by the *RS Synthesizer*. The service enables clients to request recommendations using a JSON-based model representation. The service processes such requests and sends the recommendations as a response. Section 5.3 will elaborate on this service.

Finally, in the *Client* part, any modelling tool can use the *Droid Service* to obtain recommendations and make them available to the modelling language users (label 6). Currently, our tooling supports the automatic integration of the resulting RSs within the default tree editor that EMF provides for Ecore-based languages. In Section 6.2, we will show another integration within a modelling chatbot.

### 5.2 Tool Support: The Droid Configurator

The *Droid Configurator* (https://droid-dsl.github.io/) is an Eclipse plug-in that helps the RS designer in configuring and evaluating RSs for a modelling language.

It provides a wizard where the RS designer can create *droid projects* by specifying a name for the project, the meta-model of the language for which the RS is being developed, a folder containing the models to be used for training and evaluating the RS, and the format of these models (XMI, Ecore, or UML). To simplify the RS configuration, the wizard gives the option to automatically generate a default one (i.e., default values for the "Recommendations" section in lines 30–48 of Listing 1), which the designer can modify later if so desired.

Fig. 5 shows the *Droid Configurator* environment. The Droid editor (label 1) permits the configuration of the RS via the DSL introduced in Section 4. The editor has been built using Xtext [59], and features syntax highlighting, autocompletion, and markers for errors and warnings. With label 2, the figure shows an auto-completion pop-up window to choose an existing attribute of the class Class from the UML meta-model, to serve as an item of the target class.

The environment includes a code generator that synthesizes Java code from the Droid specification. This code is in charge of evaluating the RSs. The package explorer in the figure (label 3) shows the generated Java classes in the *src-gen* folder. The RS designer does not need to look into this code, since the *RS Evaluator* component (cf. Fig. 4) automatically generates the code and displays the results in a dedicated Eclipse view (label 4).

The *Results View* (label 4) summarises in a drill-down table the evaluation results for each recommendation method and metric. The table uses different colours to facilitate the comparison of the metric values (specifically, of the values of the *F1* metric). The recommendation methods whose *F1* value is in the top 20% are shown in green; the methods whose *F1* value is under the median are shown in red; and the rest of the methods are shown in orange.

Fig. 6 shows the *Results View* in more detail. The view groups the evaluated methods by category: *Collaborative Filtering*, *Content-Based* and *Hybrid*. Within a group, each method contains a subsection per neighbourhood size, if applicable. The rows corresponding to a group show the results of the method with the best *F1* value within the group.

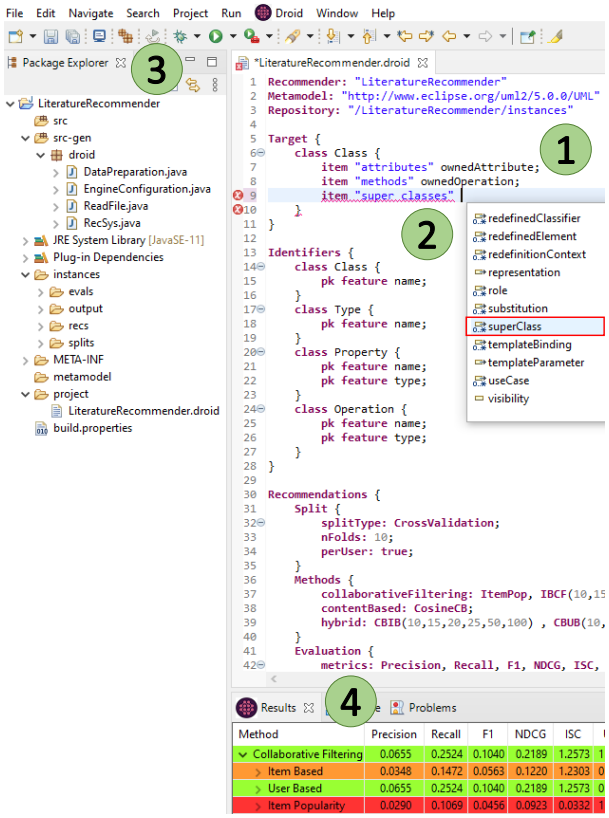



**Figure 5.** Screenshot of the *Droid Configurator*.

For example, row "Collaborative Filtering" shows the metrics of the collaborative filtering method with best *F1* value.

Results | Outline | Problems

| Method | Precision | Recall | F1 | NDCG | ISC | USC | MAP |
|---|---|---|---|---|---|---|---|
| Collaborative Filtering | 0.0655 | 0.2524 | 0.1040 | 0.2189 | 1.2573 | 1.0000 | 0.1932 |
| Item Based | 0.0348 | 0.1472 | 0.0563 | 0.1220 | 1.2303 | 0.8314 | 0.1094 |
| User Based | 0.0655 | 0.2524 | 0.1040 | 0.2189 | 1.2573 | 0.8314 | 0.1932 |
| k20 | 0.0614 | 0.2396 | 0.0978 | 0.2087 | 1.1037 | 0.8314 | 0.1835 |
| k100 | 0.0628 | 0.2415 | 0.0997 | 0.1995 | 0.7407 | 0.8314 | 0.1705 |
| k25 | 0.0642 | 0.2490 | 0.1020 | 0.2189 | 1.0228 | 0.8314 | 0.1932 |
| k50 | 0.0655 | 0.2524 | 0.1040 | 0.2147 | 0.8008 | 0.8314 | 0.1871 |
| k10 | 0.0634 | 0.2431 | 0.1005 | 0.2112 | 1.2573 | 0.7130 | 0.1861 |
| k15 | 0.0584 | 0.2292 | 0.0931 | 0.2032 | 1.1909 | 0.8195 | 0.1816 |
| Item Popularity | 0.0290 | 0.1069 | 0.0456 | 0.0923 | 0.0332 | 1.0000 | 0.0793 |
| Content Based | 0.0006 | 0.0004 | 0.0005 | 0.0005 | 0.0270 | 1.0000 | 0.0001 |
| Cosine | 0.0006 | 0.0004 | 0.0005 | 0.0005 | 0.0270 | 1.0000 | 0.0001 |
| Hybrid | 0.0456 | 0.1990 | 0.0731 | 0.1583 | 1.1411 | 1.0000 | 0.1394 |
| Content-based Item-based | 0.0031 | 0.0064 | 0.0042 | 0.0044 | 0.0207 | 0.9556 | 0.0016 |
| Content-based User-based | 0.0456 | 0.1990 | 0.0731 | 0.1583 | 1.1411 | 1.0000 | 0.1394 |

**Figure 6.** *Results View* of the *Droid Configurator*.

### 5.3 Tool Support: The Droid Service

We have built a generic recommender called DroidREST. It is a REST service implemented in Java using Jersey[1] and Tomcat[2]. The service computes the recommendations based on the configuration files generated by the *RS Synthesizer* (cf. Fig. 4). These configuration files store the trained recommender that knows which items to suggest based on the context information. Hence, there is no need to deploy a different service for each RS defined with Droid.

Clients can make POST requests to the service, which receives a recommender name together with a JSON file containing the target object of the recommendation and its context (i.e., the items that the target contains). The response to the request is a list of recommended items for the given target, using the recommendation method selected by the designer. In addition, clients can pass optional parameters for specific settings, like the maximum number of recommended items to retrieve (*newMaxRec*), the threshold for the ranking value (*threshold*), and the type of item (*itemType*). The response time of the service to calculate the recommendations is less than a second.

The REST service implementation comprises three main classes: *Recommender*, which handles the requests from clients; *ContextItem*, which parses the received JSON files to extract the recommendation target and its items from the modelling context; and *RecommenderGenerator*, which generates the recommendations for the given target taking its context and the provided query parameters into account.

### 5.4 Tool Support: Integration with EMF Tree Editor

EMF automates the synthesis of a default modelling editor starting from the Ecore meta-model of a modelling language. This editor permits creating instances of the meta-model using a tree view. Given the widespread use of these editors, our implementation generates out-of-the-box an integration of the Droid recommendation service into the default EMF tree editor of a modelling language. Next we explain the technical details of this client integration, and show an example.

In EMF, the generation of the default tree editors is automated by means of a model-to-text template language called Java Emitter Template (JET)[3]. JET supports the definition and execution of code generation templates from EMF models. This way, EMF provides a set of predefined JET templates that generate the Java code implementing the editor for a given Ecore meta-model. We have overwritten those templates to extend the generated tree editor with a "Recommender" pop-up menu on the objects that may be target of recommendations. This menu shows, for a selected object, the kinds of items that can be recommended. This information (i.e., the kinds of recommendation targets and items, see lines 5–11 in Listing 1) is not hard-coded in Java, but stored in a configuration file called *recommender.properties*. This permits building the "Recommender" menu dynamically upon clicking on an object, and facilitates the external evolution of the menu. Upon selecting a recommendation item kind for an object, a request is sent to the Droid service, passing

[1] https://eclipse-ee4j.github.io/jersey/
[2] http://tomcat.apache.org/
[3] https://projects.eclipse.org/projects/modeling.m2t.jet

the object, its context and the item type as parameters. The response is a list of recommendations, which are displayed in a table ordered by their relevance. The users can then select recommendations and apply them to the current model.

As an example, next, we show the integration of a RS specified with Droid, within the default tree editor generated for a simple modelling language for object-oriented design. The RS recommends attributes, methods and superclasses for classes. We do not use the running example to illustrate our client integration, since our approach requires starting from an Ecore meta-model and generates the whole editor from scratch, while the UML modelling editor has not been created using the JET templates predefined in EMF. The main concepts used in both examples are similar though.

Fig. 7 shows the use of the RS within the generated tree editor. The package explorer (label 1) contains a project with a model and the *recommender.properties* configuration file. The model is being edited in the window to the right (label 2). Right-clicking on any object of type Klass (label 3) shows the "Recommender" pop-up menu (label 4). This menu contains a submenu for each available kind of recommendation (in this case, "Attributes", "Methods" and "Superclasses").

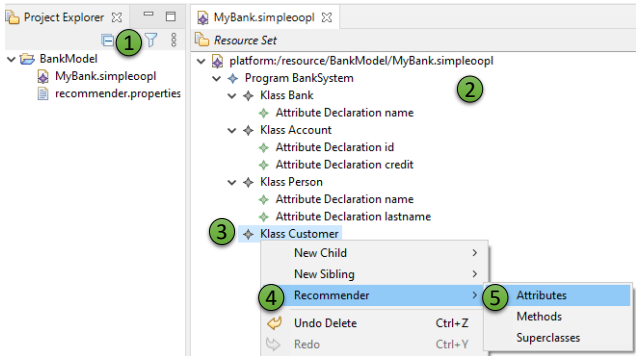



**Figure 7.** Selecting the recommendation item kind

The upper part of Fig. 8 shows the result of selecting the submenu "Attributes" on a Klass named Customer. A list of recommended attributes is presented to the user, including their name, type and rating (i.e., trust on the recommendation). When the user selects an attribute ("direction" in the figure, label 1), this is automatically added to the Klass Customer (label 2) and removed from the list.

## 6 Evaluation

With the aim to check the usefulness of the recommendations provided by Droid RSs, Section 6.1 reports on an offline evaluation with UML class models. To assess the feasibility of using Droid RSs outside Eclipse, Section 6.2 presents a case study that integrates a Droid RS with a modelling chatbot [42]. Finally, Section 6.3 discusses threats to validity.

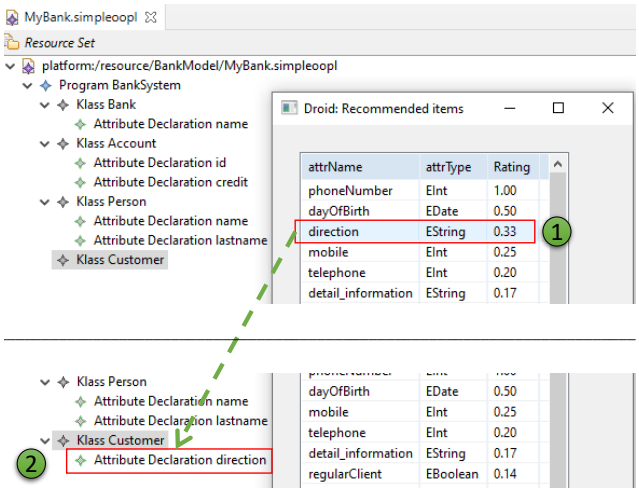



**Figure 8.** Selecting and applying a recommendation

### 6.1 Usefulness of Recommendations

The goal of this first experiment is to answer the research question (RQ) RQ1: "*How precise and complete are the recommendations provided by Droid recommenders?*". To this aim, we performed the offline experiment that is reported next.

**6.1.1 Experiment Setup.** We ran an offline experiment on two datasets from two different domains. The purpose was to analyse the performance of the RSs generated with Droid on distinct domains.

The used datasets contain models extracted from MAR [25]. This is a structure-based search engine for models and meta-models, which can be queried via input keywords. In particular, we retrieved UML models, since they are the most numerous in MAR. As domains for our experiment, we chose *Literature* and *Education*. The keywords used to retrieve the models for the *Literature* domain were *bibliography*, *book*, *author*, *journal* and *magazine*. The keywords used for the *Education* domain were *professor*, *teacher*, *student* and *alumn* (as stem of other words like *alumnus* or *alumni*). The resulting datasets are available at https://github.com/Droid-dsl/DroidConfigurator.

Table 1 shows, per each domain, the number of models, users (i.e., classes), items (i.e., attributes, methods and superclasses) and features (i.e., attributes describing users and items) in the datasets. The *Literature* and *Education* datasets have 1,447 and 1,051 UML models, respectively, conformant to the UML 2.0 class diagrams meta-model (cf. Fig. 3). The table does not consider duplicate elements. Hence, if two models contain classes with the same name, they are considered to be the same class. This is more evident in the *Education* domain, which has more models than users.

**6.1.2 Experiment.** We used Droid to configure a RS for each domain, selecting all available recommendation methods with different parameters. Specifically, we used the

**Table 1.** Description of the datasets.

| | Literature | Education |
|---|---|---|
| Num. models | 1,447 | 1,051 |
| Num. users | 1,740 | 905 |
| Num. items | 6,731 | 3,317 |
| Num. features | 6,497 | 3,231 |

Droid specification shown in Listing 1, and so, we trained multiple RSs through a variety of collaborative, content-based and hybrid recommendation methods: Item popularity (*ItemPop*), item-based collaborative filtering (*IBCF*), user-based collaborative filtering (*UBCF*), content-based with cosine similarity (*CosineCB*), content-based item-based (*CBIB*) and content-based user-based (*CBUB*). We parameterised the methods *IBCF*, *UBCF*, *CBIB* and *CBUB* with neighbourhood sizes $k$ 10, 15, 20, 25, 50 and 100. In the following, we refer to the methods that use neighbourhoods by concatenating the method name and the neighbourhood size $k$. For instance, IBCF50 refers to the IBCF $k$-NN method with 50 neighbours.

In all cases, we used *10-fold cross-validation* and a *per-user* technique to split the datasets (cf. Section 4). We analysed the performance of the RSs by means of the ranking quality metrics *precision* (p), *recall* (r), *F1*, *MAP* (Mean Average Precision) and *nDCG* (Normalized Discounted Cumulative Gain); and the coverage and diversity metrics *USC* (User Space Coverage) and *ISC* (Item Space Coverage). Additionally, in the experiment, we used a relevance threshold of 0.5, and cut-offs 5, 10, 15 and 20.

**6.1.3 Experiment Results.** Table 2 shows the results of the experiment for each domain/dataset (*Literature* and *Education*). The rows show the selected recommendation methods, and the columns correspond to the metric values. For space constraints, the table omits the results of the recommendation methods *IBCF* and *CBIB*, as their performance is worse than that of *UBCF* and *CBUB*.

We can observe that the order of magnitude of the metric values is the same in both domains. As studied in the RS field [7], this magnitude depends on many factors, such as the dataset characteristics (e.g., the average number of preferences per user, or the rating sparsity, which is the proportion of existing ratings from the whole set of potential user-item preference relations), and the evaluation methodology (e.g., the method to split training and test data, or the test ratings for which the metrics are computed). In our experiment, we followed the TestItems methodology [7] which, for a target user, evaluates recommendation lists that may contain test items from all users. This explains why the precision values are close to 0. For this reason, in general, the important aspect to consider is the relative difference of the metric values achieved by the different recommendation methods.

Analysing Table 2, a first conclusion is the fact that the content-based method *CosineCB* was the worst performing, being outperformed even by the *ItemPop* baseline. This is not surprising in our experiment. *CosineCB* estimates the preference of a user (class) for an item (attribute, method, or superclass) by means of the cosine of the angle between the user and item feature vectors. These feature vectors correspond to the names of the classes, attributes and methods in the models of the datasets. Since we do not perform any text pre-processing on those names (e.g., to unify lowercase and uppercase, singular and plural, morphological deviations, misspellings, synonyms, ambiguities), there are different names that could have been considered the same, facilitating the cosine similarity. Moreover, we may have used finer-grained user and item profiles which capture the occurrence frequency of features.

By contrast, *UBCF* and *CBUB* were the best performing recommendation methods. The results of their item-based counterparts were worse, and are not reported in the table. *UBCF* with neighbourhoods of sizes 10 and 15 achieved the best *F1* values in both domains. In terms of *MAP* and *nDCG*, which focus on the precision of the top positions in the recommendation lists, the best results were obtained with neighbourhoods of sizes 20 and 25 in the *Education* domain, and sizes 50 and 100 in the *Literature* domain. If we consider *F1*, *MAP* and *nDCG* all together, *UBCF* with neighbourhood size 15 seems the best choice for the available data and targeted task.

As expected, since *CosineCB* and *ItemPop* do not depend on user-item rating patterns, they have an *USC* of 1, which means that they are able to make recommendations for 100% of the users. In terms of *ISC* diversity, there is no significant difference between methods and domains, which reflects that both popular and unpopular items are recommended.

Table 3 shows the precision and recall of the recommendation methods on both domains per cut-off values, p@k and r@k, focusing on the first k = 5, 10, 15 and 20 recommendations. We observe that the higher the value k, the lower the precision and the higher the recall. Again, *CosineCB* was outperformed by *ItemPop*. As we explained above, the poor performance of *CosineCB* can be improved by performing some text pre-processing, which we plan to address in future work. However, even with raw data, these results are in-line with the precision reported by other RSs for class diagrams [10] (around 0.04). Although not shown in the table, *UBCF* outperformed *IBCF*. The hybrid use of content-based and collaborative filtering techniques did not improve the recommenders based on a single technique. When considering both p@k and r@k, *UBCF* with neighbourhood size 50 was the best performing method.

Answering RQ1, our evaluation shows that standard recommendation methods are able to provide sensible recommendations for every class, starting from relatively small datasets that have not been pre-processed. These results are in-line with RSs specifically created for class diagrams [10]. Still, we have identified some aspects that would allow

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

| Method | Literature | | | | | | | Education | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | p | r | F1 | MAP | nDCG | USC | ISC | p | r | F1 | MAP | nDCG | USC | ISC |
| ItemPop | 0.006 | 0.180 | 0.012 | 0.055 | 0.086 | **1.000** | 0.012 | 0.007 | 0.224 | 0.013 | 0.082 | 0.117 | **1.000** | 0.017 |
| CosineCB | 0.001 | 0.032 | 0.002 | 0.017 | 0.020 | **1.000** | 0.004 | 0.002 | 0.076 | 0.004 | 0.003 | 0.017 | **1.000** | 0.007 |
| UBCF10 | **0.033** | 0.290 | **0.060** | 0.157 | 0.195 | 0.824 | 0.059 | **0.035** | 0.337 | **0.064** | 0.184 | 0.227 | 0.830 | 0.056 |
| UBCF15 | **0.026** | 0.304 | **0.048** | 0.160 | 0.201 | 0.860 | **0.060** | **0.026** | 0.346 | **0.048** | **0.184** | 0.228 | 0.863 | **0.057** |
| UBCF20 | 0.022 | 0.319 | 0.041 | **0.161** | 0.205 | 0.863 | **0.060** | 0.022 | 0.360 | 0.042 | 0.183 | **0.232** | 0.868 | **0.057** |
| UBCF25 | 0.020 | 0.327 | 0.038 | **0.163** | 0.208 | 0.865 | **0.060** | 0.021 | 0.369 | 0.039 | **0.184** | **0.235** | 0.868 | **0.057** |
| UBCF50 | 0.019 | **0.348** | 0.037 | 0.159 | **0.211** | 0.865 | 0.058 | 0.018 | **0.383** | 0.035 | 0.176 | 0.231 | 0.868 | **0.057** |
| UBCF100 | 0.020 | **0.360** | 0.037 | 0.155 | **0.210** | 0.865 | 0.056 | 0.019 | **0.387** | 0.035 | 0.166 | 0.224 | 0.868 | 0.055 |
| CBUB10 | 0.015 | 0.202 | 0.028 | 0.108 | 0.132 | 0.929 | 0.055 | 0.023 | 0.258 | 0.042 | 0.137 | 0.168 | 0.926 | 0.053 |
| CBUB15 | 0.011 | 0.210 | 0.022 | 0.105 | 0.130 | 0.962 | 0.056 | 0.015 | 0.260 | 0.029 | 0.135 | 0.165 | 0.984 | 0.054 |
| CBUB20 | 0.009 | 0.213 | 0.017 | 0.102 | 0.129 | 0.963 | 0.056 | 0.012 | 0.265 | 0.022 | 0.133 | 0.165 | **1.000** | 0.055 |
| CBUB25 | 0.008 | 0.212 | 0.015 | 0.098 | 0.125 | 0.987 | 0.056 | 0.010 | 0.271 | 0.019 | 0.133 | 0.166 | **1.000** | 0.055 |
| CBUB50 | 0.006 | 0.212 | 0.011 | 0.088 | 0.117 | **1.000** | 0.055 | 0.008 | 0.304 | 0.016 | 0.133 | 0.176 | **1.000** | 0.055 |
| CBUB100 | 0.007 | 0.242 | 0.014 | 0.097 | 0.133 | **1.000** | 0.051 | 0.008 | 0.302 | 0.016 | 0.124 | 0.169 | **1.000** | 0.052 |

**Table 3.** Results of the experiment per cut-offs. The best values are shown in bold.

| Method | Literature | | | | | | | | Education | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | p@5 | p@10 | p@15 | p@20 | r@5 | r@10 | r@15 | r@20 | p@5 | p@10 | p@15 | p@20 | r@5 | r@10 | r@15 | r@20 |
| ItemPop | 0.022 | 0.014 | 0.012 | 0.010 | 0.072 | 0.090 | 0.110 | 0.123 | 0.027 | 0.018 | 0.015 | 0.012 | 0.099 | 0.129 | 0.154 | 0.168 |
| CosineCB | 0.003 | 0.002 | 0.002 | 0.001 | 0.016 | 0.018 | 0.020 | 0.021 | 0.001 | 0.001 | 0.001 | 0.001 | 0.001 | 0.004 | 0.006 | 0.007 |
| UBCF10 | 0.053 | 0.033 | 0.025 | 0.020 | 0.194 | 0.231 | 0.253 | 0.265 | 0.061 | 0.037 | 0.027 | 0.022 | 0.228 | 0.270 | 0.295 | 0.308 |
| UBCF15 | 0.055 | 0.034 | 0.026 | 0.021 | 0.200 | 0.237 | 0.259 | 0.272 | 0.060 | 0.038 | 0.028 | 0.022 | 0.227 | 0.272 | 0.299 | 0.313 |
| UBCF20 | 0.057 | 0.036 | 0.027 | 0.022 | 0.205 | 0.245 | 0.267 | 0.282 | **0.062** | 0.038 | 0.029 | 0.023 | **0.232** | 0.276 | 0.305 | 0.320 |
| UBCF25 | 0.058 | 0.037 | 0.028 | 0.022 | 0.207 | 0.250 | 0.274 | 0.289 | **0.063** | **0.039** | **0.029** | 0.023 | **0.235** | **0.281** | **0.310** | **0.329** |
| UBCF50 | **0.060** | **0.038** | **0.029** | **0.023** | **0.212** | **0.261** | **0.286** | **0.302** | 0.062 | **0.039** | **0.030** | **0.024** | 0.229 | **0.279** | **0.313** | **0.335** |
| UBCF100 | **0.060** | **0.039** | **0.029** | **0.024** | **0.210** | **0.260** | **0.287** | **0.306** | 0.059 | 0.038 | 0.029 | **0.024** | 0.217 | 0.270 | 0.302 | 0.327 |
| CBUB10 | 0.031 | 0.018 | 0.013 | 0.011 | 0.139 | 0.163 | 0.176 | 0.184 | 0.040 | 0.024 | 0.018 | 0.014 | 0.178 | 0.209 | 0.226 | 0.237 |
| CBUB15 | 0.030 | 0.019 | 0.014 | 0.011 | 0.135 | 0.165 | 0.180 | 0.188 | 0.039 | 0.024 | 0.018 | 0.014 | 0.174 | 0.206 | 0.226 | 0.236 |
| CBUB20 | 0.029 | 0.019 | 0.014 | 0.011 | 0.130 | 0.168 | 0.184 | 0.193 | 0.038 | 0.024 | 0.018 | 0.014 | 0.170 | 0.205 | 0.225 | 0.236 |
| CBUB25 | 0.028 | 0.018 | 0.014 | 0.011 | 0.124 | 0.162 | 0.180 | 0.190 | 0.038 | 0.024 | 0.018 | 0.014 | 0.169 | 0.206 | 0.227 | 0.239 |
| CBUB50 | 0.025 | 0.016 | 0.013 | 0.011 | 0.114 | 0.142 | 0.167 | 0.183 | 0.044 | 0.027 | 0.021 | 0.017 | 0.173 | 0.211 | 0.243 | 0.261 |
| CBUB100 | 0.034 | 0.022 | 0.017 | 0.014 | 0.129 | 0.163 | 0.182 | 0.197 | 0.043 | 0.026 | 0.021 | 0.017 | 0.166 | 0.202 | 0.230 | 0.252 |

improving the generated recommendations, such as using larger datasets, pre-processing the text features that the content-based methods exploit, or even incorporating more specific, task-oriented recommendation methods.

### 6.2 Case Study on RS Integration

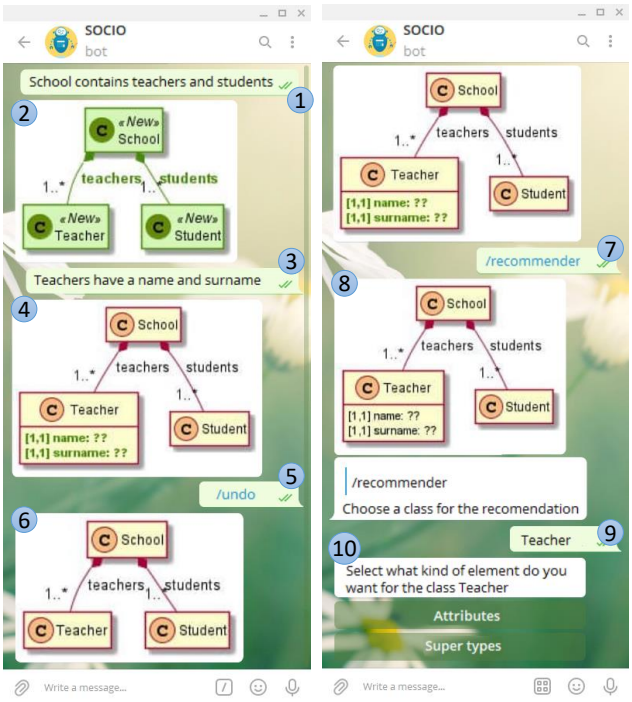
This section shows a case study on the integration of a RS specified with Droid into a modelling chatbot called Socio. With this study, we aim to answer the following RQ (RQ2): *How difficult is it to integrate a Droid-based RS with a non-Eclipse-based modelling client?*

Socio [42] is a chatbot or conversational agent that enables heterogeneous groups of domain and modelling experts to collaborate on modelling tasks. It works in social networks, like Telegram or Twitter, and facilitates the active participation of domain experts with no technical background in building models (class diagrams) by using natural language (NL) as the modelling interface.

Fig. 9(a) shows a user interaction with Socio in Telegram. The user can send messages expressing domain requirements in NL to the chatbot (labels 1 and 3). Socio interprets the messages and the current status of the model, infers the necessary modelling actions, updates the model, and sends back an image of the model with the modified elements in green (labels 2 and 4). For example, given the message "School contains teachers and students" (label 1), Socio infers that there must be three classes named *School*, *Teacher* and *Student*. Then, because of the *contains* verb, it infers that *School* should have two containment references with cardinality one to many (as teachers and students are plural), one called *teachers* and going to *Teacher*, and the other called *students* and going to *Student*. Since the model is empty at this moment, Socio creates all these elements (label 2).

Users normally do not provide all requirements in a single message, and so, Socio permits a model to be incomplete or incorrect. The interaction with label 3 illustrates this. When the user says "Teachers have a name and surname", Socio interprets that there must be a class named *Teacher* with two features, *name* and *surname*. Since the class already exists, it only adds the two features, but since there is no information about their types, their definition is incomplete (label 4).

Besides model creation via NL processing, the chatbot has commands to manage, validate, download the model, or undo and redo the modelling actions. In Telegram, these commands start by a backslash followed by a keyword. Labels 5 and 6 in Fig. 9(a) show an example of the undo command.




a) SOCIO modelling examples b) Recommender command

**Figure 9.** Example of Socio interaction in Telegram.

For this case study, we extended Socio with a RS specified with Droid and hence available as a service. Fig. 10 shows a scheme of the integration of the RS within Socio, where the new components are highlighted in green. Socio has a front-end provided by Telegram and a back-end. The latter is the main component of the architecture since it handles all the functionality of Socio: Information and model storage, NL processing and modelling actions. The Telegram client connects the user interaction in Telegram with the back-end.

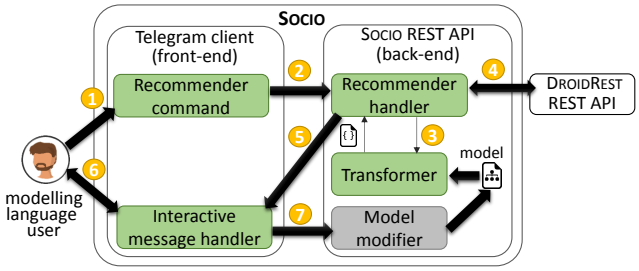



**Figure 10.** Architecture of Droid integration with Socio

For the integration, we created a Recommender command on the client side (label 1 in Fig. 10). When the user types this command to obtain recommendations (label 1), the Telegram client sends a request to the back-end, which is handled by the *Recommender handler* (label 2). Since the model is internally represented with EMF, a *Transformer* converts the context element of the recommendation into the JSON format required by Droid (label 3). Then, the *Recommender handler* requests recommendations to the Droid service (label 4), and sends the returned recommendations back to the client (label 5). In the client, an *Interactive message handler* transforms the recommendations into an interactive message containing one button per recommended item (label 6). When the user selects one of these buttons, the handler sends a request to the back-end to add the selected item to the model (label 7). Then, the selected button is deleted, while the other buttons remain available to permit applying further recommendations.

Fig. 9(b) shows the usage of the /recommender command in Telegram. When a user types the command (label 7), Socio displays the current model and prompts the user to select a class (label 8). Once the user selects a class (label 9), Socio asks the kind of items to be recommended (label 10). Since Socio models do no support methods, the user can choose the recommendation of attributes and supertypes.

Fig. 11 illustrates the recommendations provided by Droid. It shows the recommended supertypes (label 1) and attributes (label 2) for the class *Teacher*. When the user presses the button with the recommendation *Person*, Socio creates a new class because it does not exist, and adds it as a supertype of *Teacher*. When the user presses the button with the recommendation *name*, Socio detects that *Teacher* already defines this attribute and only updates its type. This way, recommendations not only add new elements to the model, but sometimes also allow fixing incomplete elements.

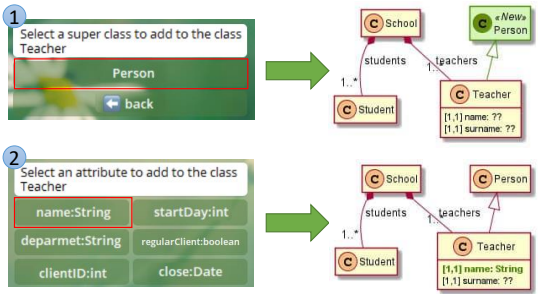



**Figure 11.** Droid recommendations in Socio.

Table 4 shows the LOC and number of Java classes developed to achieve the RS integration. The *Interactive message handler* is the largest component, which is normal as it handles several user interactions. We can observe that the integration did not require many changes in the Socio architecture, and the new components are not large.

Answering RQ2, this case study proves that Droid-based RSs can be easily integrated with tools outside Eclipse. While the integration with Socio has not many LOC, we added code on both its front-end and its back-end. Moreover, more than 50% of the code was dedicated to the user interaction.

**Table 4.** Metrics for integrating Droid with Socio.

| | | LOC | Num. Classes |
|---|---|---|---|
| Back | Recommender handler | 160 | 2 |
| | Transformer | 44 | 1 |
| Front | Recommender command | 128 | 2 |
| | Interac. message handler | 400 | 1 |
| | Total | 732 | 6 |

These two circumstances can make a big difference in the effort required to integrate the RS with other modelling tools.

### 6.3 Threats to Validity

Next, we discuss threats to validity in our evaluations.

With respect to the offline experiment of Section 6.1, we tried to minimize the threats to its external validity by making use of two independent and large datasets from two different domains. However, the domain selection and the keywords chosen for the query may affect the generality of the results. To tackle this issue, we plan to conduct experiments with other domains and datasets in the future. Another threat is the specific recommendation task accomplished in the experiment, namely the completion of class diagrams with new attributes, methods and superclasses. Hence, further experiments are needed to draw conclusions on the use of our approach for other recommendation tasks. Concerning internal validity, we tried to avoid any bias on the results by the use of third-party datasets.

Regarding the case study reported in Section 6.2, the results are specific to Socio and cannot be generalized, which remains the main threat to the external validity. However, the integration of Socio was specially challenging due to its distributed architecture and its independence of Eclipse; hence, we expect that the effort to integrate a Droid-based RS in other clients will not be higher than for Socio.

## 7 Related Work

In this section, we review the two main areas of related works: Recommenders for modelling languages, and automated approaches for the synthesis of RSs.

### 7.1 Recommenders for Modelling Languages

According to [4], the most common usage purposes for recommenders in MDE are completion, finding, repair, reuse, and to a lesser extent, creation of modelling artefacts. The recommendations typically apply to models and meta-models, while recommenders for model transformations and code generators are scarce. Droid can be applied to any kind of artefact, provided that it is defined by a meta-model.

Most recommenders for modelling languages target UML, especially class diagrams. IPSE [20] has a knowledge-based RS that guides students on creating class diagrams, and the recommendations build on Prolog constraints defined by the teacher. RapMOD [31] recommends relevant autocompletion actions for graphical UML class diagrams. REBUILDER [23] relies on case-based reasoning, Bayesian networks and WordNet to recommend class diagrams similar to a given one. Elkamel et al. [16] use similarity metrics to recommend similar classes to the ones in the current class diagram. Other researchers propose RSs for other UML diagrams: Cerqueira et al. [12] propose a CB approach for recommending behavioural features for UML sequence diagrams, and Aquino et al. [5] present a recommender of actors and use cases for use case diagrams. While these works tackle useful modelling tasks, they serve a specific modelling language and the recommendation method is fixed. Instead, Droid is not UML-specific but it permits customizing the target modelling language, the kind of items to be recommended, and the recommendation algorithm.

Some approaches aim to provide semantically related terms and context-sensitive information for a modelling task. Burgueño et al. [10] propose a domain concept recommender based on the analysis of the textual information available on the domain model being constructed, as well as on general knowledge about the business domain. The domain modelling tool DoMoRe [2] exploits a knowledge base of domain-specific terms and their relationships to provide context-sensitive recommendations. Other tools, like Extremo [35] or the assistant envisioned by Savary-Leblanc [51], employ semantic similarity based on lexical databases like WordNet to recommend semantically related terms. While these tools target a specific modelling task, our framework is generic and configurable for arbitrary modelling languages.

Recommenders have also been applied to business process modelling. For example, to recommend complete process models based on the user profile [28], as well as finer-grained recommendations that pursue completing a process model with new fragments [30], activity nodes [14, 32], tasks [44] or actor roles [44]. Again, these works are specific to a modelling language, and the recommendation method is fixed.

In contrast to the previous language-specific approaches, others are language-independent. These are typically applicable to arbitrary modelling languages defined in a given meta-modelling framework, such as EMF. For example, PARMOREL [6, 26] uses reinforcement learning to repair malformed EMF models based on the user preferences and the experience gained from previous repairs. ReVision [41] suggests consistency-preserving model editing rules for model repair. SimVMA [54] uses clone detection to help modellers find models or operations relevant to them. Finally, Kögel [29] proposes to analyse the history of past model changes to suggest recommendations, and foresees the use of machine learning, heuristic search algorithms, association rules and decision trees. Altogether, even though these works plan on frameworks for different languages, the recommendation method is fixed, and the recommendations cannot be customised, as we can do using Droid.

### 7.2 Recommender System Generation

While we can find many RSs for modelling languages, most were developed by hand from scratch, which requires a high effort [36]. Hence, recent studies [4] have identified the need of methods and tools automating the construction of recommenders for modelling languages. This work aims to fill this gap. Next, we compare with other related approaches.

Fellmann et al. [19] define a reference model with the data perspective requirements of RSs for process modelling. The model can be instantiated as a guide for developing new process modelling recommenders, or to assess existing ones. While useful, the approach is specific to process modelling, and does not provide automation or code synthesis.

Rojas et al. [48] present an MDE framework to create mobile RSs of geographic points of interest. The framework helps defining the structural, behavioural and navigational aspects of the RS, and customising the user preferences, similarity metrics and similarity formula. In [47], a similar solution is used to recommend trips and tours. However, in both works, the target domain of the recommendation is fixed.

We also find MDE proposals to support non-expert users on applying data mining. For example, Espinosa at al. [17, 18] reuse the past experiences of data mining experts to compute the accuracy for a given new dataset and recommend the one with the best performance. The framework permits customising the data mining task to perform, the evaluation method and metrics, and the mining algorithm. Even though this solution offers the flexibility and benefits of MDE, the generated recommenders are data mining applications.

In a more general setting, Hermes [15] is a generic framework to build recommenders for modelling environments. Its extensible architecture permits defining new recommendation strategies, new widgets to trigger and display the recommendations, and new contexts to adapt the recommendations to the modelling environment. These elements are coded as extensions of base classes, or registered in the case of resources like icons and labels. Hermes provides a dashboard to define the class extensions, and supports the manual testing of the recommender. In contrast, our DSL Droid does not require coding, but it provides a simple syntax to configure the kinds of recommended items, the recommendation method, and its evaluation based on standard metrics. Moreover, it automatically generates a tailored RS as a web service to make it available from arbitrary environments.

More similar to our proposal, the vision paper [52] foresees a lowcode development environment where end users can define RSs by using graphical interfaces, drag-and-drop utilities and forms. The authors aim to support the construction of arbitrary RSs, not specific for modelling languages. The lowcode environment will build on a generic meta-model to provide components implementing recurring functionalities for RSs, such as data pre-processing, capturing context, and producing and presenting recommendations. The authors foresee having several DSLs to configure each aspect of the recommender. Our philosophy is similar, but we focus on RSs for modelling. This way, our DSL allows the fine-grained specification of the recommendation target and items, and our tooling generates a RS available as a REST API that can be integrated in other tools.

## 8 Conclusions and Future Work

RSs are increasingly being used in Software Engineering, and MDE is no exception to this trend. Since building RSs for DSLs is time expensive, we have developed a model-based approach to automate their construction. The approach provides a DSL to configure the target of the recommendation and the type of the recommended items, and supports the evaluation of the RSs to identify the best one for the problem at hand. Our solution relies on a generic recommendation service that can be integrated out-of-the-box with the EMF tree editor for models. We have demonstrated the feasibility of its integration with non-Eclipse tools, and have evaluated the precision and completeness of the recommendations.

In practice, the creation of RSs for modelling requires having big sets of models for training. These exist for popular modelling languages (e.g., UML, BPMN, Simulink), but not for other DSLs. We trust that the emergence of dedicated model search engines [25] and repositories will facilitate this task. Moreover, there are other options. First, RSs can be trained with the available models, and retrained as more models become available. Second, one may apply "transfer learning" for some DSLs, i.e., training the RS with models of another similar DSL. For example, one may build a RS for UML class diagrams, and apply it to Ecore meta-models.

In the future, we plan to work on pre-processing techniques for the model sets. For instance, for the UML class recommender, it could be useful to pre-process names (e.g., deleting blank spaces) and cluster semantically similar names. We would also like to enrich the recommendation context, e.g., including classes related to the recommendation target. We have focussed on classical recommendation methods, but we aim to make Droid a DSL front-end to configure arbitrary recommendation methods, including those specific for modelling tasks. For this purpose, we are making our architecture extensible via extension points, to be implemented for specific methods. Another line to explore is to gather recommendation feedback from the users, and adjust future recommendations based on it. Finally, we plan to make a user study to identify strengths and weaknesses of our proposal.

## 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 Transactions on Knowledge and Data Engineering* 17, 6 (2005), 734–749.

[2] Henning Agt-Rickauer, Ralf-Detlef Kutsche, and Harald Sack. 2018. DoMoRe - A recommender system for domain modeling. In *6th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. SciTePress, 71–82.

[3] Lissette Almonte, Iván Cantador, Esther Guerra, and Juan de Lara. 2020. Towards automating the construction of recommender systems for low-code development platforms. In *Proc MODELS Companion Proceedings*. ACM, 66:1–66:10.

[4] Lissette Almonte, Esther Guerra, Iván Cantador, and Juan de Lara. 2021. Recommender systems in model-driven engineering: A systematic mapping review. *Software and System Modeling* in press (2021).

[5] Erika Rizzo Aquino, Pierre de Saqui-Sannes, and Rob A. Vingerhoeds. 2020. A methodological assistant for use case diagrams. In *8th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. SciTePress, 227–236.

[6] Angela Barriga, Adrian Rutle, and Rogardt Heldal. 2020. Improving model repair through experience sharing. *Journal of Object Technology* 19, 2 (2020), 13:1–21.

[7] Alejandro Bellogín, Iván Cantador, and Pablo Castells. 2013. A comparative study of heterogeneous item recommendations in social systems. *Information Sciences* 221 (2013), 142–169.

[8] Markus Borg, Krzysztof Wnuk, Björn Regnell, and Per Runeson. 2017. Supporting change impact analysis using a recommendation system: An industrial case study in a safety-critical context. *IEEE Transactions on Software Engineering* 43, 7 (2017), 675–700.

[9] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2017. *Model-Driven Software Engineering in Practice, Second Edition*. Morgan & Claypool Publishers, San Rafael, California (USA).

[10] Loli Burgueño, Robert Clarisó, Shuai Li, Sébastien Gérard, and Jordi Cabot. 2021. An NLP-based architecture for the autocompletion of partial domain models. In *CAiSE (LNCS, Vol. 12751')*. Springer International Publishing, 91–106.

[11] Robin Burke. 2002. Hybrid recommender systems: Survey and experiments. *User Modeling and User-adapted Interaction* 12, 4 (2002), 331–370.

[12] Thaciana Cerqueira, Franklin Ramalho, and Leandro Balby Marinho. 2016. A content-based approach for recommending UML sequence diagrams. In *28th International Conference on Software Engineering and Knowledge Engineering (SEKE)*. KSI Research Inc. and Knowledge Systems Institute Graduate School, 644–649.

[13] Marcos César de Oliveira, Davi Freitas, Rodrigo Bonifácio, Gustavo Pinto, and David Lo. 2019. Finding needles in a haystack: Leveraging co-change dependencies to recommend refactorings. *Journal of Systems and Software* 158 (2019).

[14] ShuiGuang Deng, Dongjing Wang, Ying Li, Bin Cao, Jianwei Yin, Zhaohui Wu, and Mengchu Zhou. 2017. A recommendation system to facilitate business process modeling. *IEEE Transactions on Cybernetics* 47, 6 (2017), 1380–1394.

[15] Andrej Dyck, Andreas Ganser, and Horst Lichter. 2014. A framework for model recommenders - Requirements, architecture and tool support. In *2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. SciTePress, 282–290.

[16] Akil Elkamel, Mariem Gzara, and Hanêne Ben-Abdallah. 2016. An UML class recommender system for software design. In *13th IEEE/ACS International Conference of Computer Systems and Applications (AICCSA)*. IEEE Computer Society, 1–8.

[17] 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.

[18] 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.

[19] Michael Fellmann, Dirk Metzger, Sven Jannaber, Novica Zarvic, and Oliver Thomas. 2018. Process modeling recommender systems - A generic data model and its application to a smart glasses-based modeling environment. *Bus. Inf. Syst. Eng.* 60, 1 (2018), 21–38.

[20] H. Garbe. 2012. Intelligent assistance in a problem solving environment for UML class diagrams by combining a generative system with constraints. In *eLearning*. IADIS, 412–416.

[21] Marko Gasparic and Andrea Janes. 2016. What recommendation systems for software engineering recommend: A systematic literature review. *Journal of Systems and Software* 113 (2016), 101–113. https://doi.org/10.1016/j.jss.2015.11.036

[22] Github. 2021. Copilot. https://copilot.github.com/.

[23] Paulo Gomes. 2004. Software design retrieval using Bayesian networks and WordNet. In *7th European Conf. on Advances in Case-Based Reasoning (ECCBR) (Lecture Notes in Computer Science, Vol. 3155)*. Springer, 184–197.

[24] Asela Gunawardana and Guy Shani. 2015. Evaluating recommender systems. In *Recommender Systems Handbook*. Springer, 265–308.

[25] José Antonio Hernández López and Jesús Sánchez Cuadrado. 2020. MAR: a structure-based search engine for models. In *MoDELS '20*. ACM, 57–67.

[26] Ludovico Iovino, Angela Barriga, Adrian Rutle, and Rogardt Heldal. 2020. Model repair with quality-based reinforcement learning. *Journal of Object Technology* 19, 2 (2020), 17:1–21.

[27] Steven Kelly and Juha-Pekka Tolvanen. 2008. *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley.

[28] Hadjer Khider, Slimane Hammoudi, and Abdelkrim Meziane. 2020. Business process model recommendation as a transformation process in MDE: Conceptualization and first experiments. In *8th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. SciTePress, 65–75.

[29] Stefan Kögel. 2017. Recommender system for model driven software development. In *11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 1026–1029.

[30] Agnes Koschmider, Thomas Hornung, and Andreas Oberweis. 2011. Recommendation-based editor for business process modeling. *Data & Knowledge Engineering* 70, 6 (2011), 483–503.

[31] Tobias Kuschke and Patrick Mäder. 2017. RapMOD - in situ autocompletion for graphical models: poster. In *39th International Conference on Software Engineering (ICSE), Companion Volume*. IEEE Computer Society, 303–304.

[32] Ying Li, Bin Cao, Lida Xu, Jianwei Yin, ShuiGuang Deng, Yuyu Yin, and Zhaohui Wu. 2014. An efficient recommendation method for improving business process modeling. *IEEE Transactions on Industrial Informatics* 10, 1 (2014), 502–513.

[33] Pasquale Lops, Marco De Gemmis, and Giovanni Semeraro. 2011. Content-based recommender systems: State of the art and trends. In *Recommender Systems Handbook*. Springer, 73–105.

[34] Pyry Matikainen, P. Michael Furlong, Rahul Sukthankar, and Martial Hebert. 2013. Multi-armed recommendation bandits for selecting state machine policies for robotic systems. In *2013 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 4545–4551.

[35] Ángel Mora Segura, Juan de Lara, Patrick Neubauer, and Manuel Wimmer. 2018. Automated modelling assistance by integrating heterogeneous information sources. *Computer Languages, Systems and Structures* 53 (2018), 90–120.

[36] Gunter Mussbacher, Benoît Combemale, Jörg Kienzle, Silvia Abrahão, Hyacinth Ali, Nelly Bencomo, Márton Búr, Loli Burgueño, Gregor Engels, Pierre Jeanjean, Jean-Marc Jézéquel, Thomas Kühn, Sébastien

Mosser, Houari A. Sahraoui, Eugene Syriani, Dániel Varró, and Martin Weyssow. 2020. Opportunities in intelligent modeling assistance. *Softw. Syst. Model.* 19, 5 (2020), 1045–1053.

[37] 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 *41st International Conference on Software Engineering (ICSE)*. IEEE / ACM, 1050–1060.

[38] Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, and Massimiliano Di Penta. 2020. CrossRec: Supporting software developers by recommending third-party libraries. *Journal of Systems and Software* 161 (2020).

[39] Xia Ning, Christian Desrosiers, and George Karypis. 2015. A comprehensive survey of neighborhood-based recommendation methods. In *Recommender Systems Handbook*. Springer, 37–76.

[40] OCL. 2014. http://www.omg.org/spec/OCL/.

[41] Manuel Ohrndorf, Christopher Pietsch, Udo Kelter, and Timo Kehrer. 2018. ReVision: a tool for history-based model repair recommendations. In *40th International Conference on Software Engineering (ICSE), Companion Proceeedings*. ACM, 105–108.

[42] Sara Pérez-Soler, Esther Guerra, and Juan de Lara. 2018. Collaborative modeling and group decision making using chatbots in social networks. *IEEE Softw.* 35, 6 (2018), 48–54.

[43] Ana Pescador and Juan de Lara. 2016. DSL-maps: from requirements to design of domain-specific languages. In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 438–443. https://doi.org/10.1145/2970276.2970328

[44] Mohammad Ehson Rangiha, Marco Comuzzi, and Bill Karakostas. 2015. Role and task recommendation and social tagging to enable social business process management. In *BPMDS/EMMSAD@CAiSE (Lecture Notes in Business Information Processing, Vol. 214)*. Springer, 68–82.

[45] Z. Reitermanová. 2010. Data splitting. In *WDS*. Matfyzpress, 31–36.

[46] Martin P. Robillard, Robert J. Walker, and Thomas Zimmermann. 2010. Recommendation systems for Software Engineering. *IEEE Software* 27, 4 (2010), 80–86.

[47] 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.

[48] 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.

[49] Alan Said and Alejandro Bellogín. 2014. Rival: a toolkit to foster reproducibility in recommender system evaluation. In *Eighth ACM Conference on Recommender Systems, RecSys '14*. ACM, 371–372. See also https://github.com/recommenders/rival.

[50] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. 2018. Quick fixing ATL transformations with speculative analysis. *Software and Systems Modeling* 17, 3 (2018), 779–813.

[51] Maxime Savary-Leblanc. 2019. Improving MBSE tools UX with AI-empowered software assistants. In *22nd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS), Companion Volume*. IEEE, 648–652.

[52] Claudio Di Sipio, Davide Di Ruscio, and Phuong T. Nguyen. 2020. Democratizing the development of recommender systems by means of low-code platforms. In *MODELS '20: ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems*, Esther Guerra and Ludovico Iovino (Eds.). ACM, 68:1–68:9.

[53] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2008. *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Professional, Upper Saddle River, NJ.

[54] Matthew Stephan. 2019. Towards a cognizant virtual software modeling assistant using model clones. In *41st International Conference on Software Engineering: New Ideas and Emerging Results (NIER@ICSE)*. IEEE / ACM, 21–24.

[55] Masateru Tsunoda, Takeshi Kakimoto, Naoki Ohsugi, Akito Monden, and Ken-ichi Matsumoto. 2005. Javawock: A Java class recommender system based on collaborative filtering. In *17th International Conference on Software Engineering and Knowledge Engineering (SEKE)*. 491–497.

[56] UML. 2017. UML 2.5.1 OMG specification. http://www.omg.org/spec/UML/2.5.1/.

[57] Saúl Vargas and Pablo Castells. 2011. Rank and relevance in novelty and diversity metrics for recommender systems. In *Fifth ACM Conference on Recommender Systems, RecSys '11*. ACM, New York, NY, USA, 109–116. See also http://ranksys.github.io/.

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

[59] Xtext. 2021. http://www.eclipse.org/Xtext/. (last accessed in July 2021).