Enabling Model Recommenders for Command-Enabled Editors

Andrej Dyck, Andreas Ganser, and Horst Lichter

RWTH Aachen University, Ahornstr. 55, 52074 Aachen, Germany
{dyck, ganser, lichter}@swc.rwth-aachen.de,
home page: http://www.swc.rwth-aachen.de

Abstract. Content assist systems and code completion are nicely accessible in integrated development environments (IDEs). Using multiple data sources and performing sophisticated completion in several editors is quite common. However, no such supporting system exists for modeling environments, e.g., a completion mechanism in class diagrams is only existent for textual items like names, if at all.
We designed a framework to bolster model recommendation research and briefly present the architecture and the realization in this paper. Both are easily extendable via hot spots by new data recommendation strategies or by completely new environments like editors. As additional tool support for extending this framework, we provide a dashboard, which eases initial development for new extensions. Accordingly, researchers get all the conceptual groundwork and an implemented infrastructure explained in a tutorial manner that eases the initial burden to get recommendations going for modeling environments. These could produce recommendations from various sets of data, e.g., example models, patterns, best practices, or template enhanced models.

Keywords: Generic Recommendation Framework, Recommender Systems, Model Completion, Modeling Support Model Recommendation, Software Framework, Model Reuse, MDE, MDD, EMF

1 Intro

Todays web shops try to simulate good salesmen by providing recommendations that could be of interest for a customer. They do so by learning shopping behavior and using this data to produce recommendations to customers. For example, NetFlix, Amazon, and other online web shops do so, build user profiles, and recommend items that "other customers also bought" (Amazon). But these recommendations have limitations and should sometimes be seen as examples of what the customer actually wants, because some attributes of the products might be to specific. For example, a shirt of a certain brand might be the right recommendation, but the system might not be able to predict the proper size or wanted color of the shirt.

Now, the ideas from recommender systems proved so beneficial for web shops that other domains try to apply these approaches as well. For example, recommender systems for software engineering (RSSE) try to integrate these ideas in engineering processes of software [Maalej et al., 2013]. So far, most of the research was conducted in the area of auto complete functionality for integrated development environments. For example, the Code Recommender plugin for Eclipse enhances the Java Development Tools code completion with more sophisticated recommendations compared to types, methods, and identifiers.

We believe that another domain that could hugely benefit from recommender system support is modeling environments. At first, recommendations in modeling environments appear similar to the ones in programming, but there are differences. Considering UML class diagrams, these are due to the individual scope of modeling. Hence, a recommendation comprising several classes is very unlikely to be the perfect match of what a modeler needs and should be rather seen as an eighty percent solution. Moreover, often a variety of valid solutions exists, since modeling usually is a matter of subjectivity. Still, producing recommendations from a repository of indexed examples could offer different possible solutions.

The limitations that might be put on producing recommendations for class diagrams are in the early stages of research and it proved helpful to separate between three areas. First, the content itself limits the kind of recommendations, because depending on the data source, a name or an entire model might be recommended. Second, the current context influences how appropriate recommendations are, e.g., editing the name field of a class should limit recommendations to textual recommendations only; recommending an interface or a type would not help. Third, the user interface (UI) restricts how recommendations are presented since pro-active and re-active systems work differently.

Therefore, we contribute a research environment explained in a tutorial manner, hoping to bolster further research by presenting our requirements analysis for such an environment (section 3.1), an architecture (section 3.2), some more insights (section 3.3), and dashboard support (section 4).

2 Related Work

Recommender Systems could be traced back to Information Management Systems and Decision Supporting Systems (DSS) in the eighties [Sprague, 1980]. But we keep to the recent terminology and adhere to the conceptual shift, looking into more recent frameworks without contrasting them to DSS.

White et al. present a framework for domain specific modeling languages on a conceptual level [White and Schmidt, 2006]. They focus on establishing domain specific knowledge bases and algorithm so they can work in what they call “combinatorically challenging domains”. The foundation of their solution is in Prolog and they demonstrated their system with an example modeled in AUTOSAR. In contrast, our framework does not focus on editors or domains but provides a conceptual and implemented infrastructure, so their implementation could be plugged into our solution among others.

Prolog is also the bases of Sen et al.'s approach [Sen et al., 2008]. They demonstrate their "partial model completion" with finite state machines and offer a brief methodological overview as well. Our work differs in respect that we do not focus on graphical aspects for domain specific graphical editors but rather on a higher level of management. Hence, their solution could be plugged into our framework as another extension, if their architecture was suitable.

Moreover, tripple graph grammars are the foundation of the approach by Mazanek et al. [Mazanek et al., 2008]. They work on Nassi-Shneiderman diagrams and transform them into graph grammars, which they can leverage for auto complete functionality. In this respect they produce suggestions which could be called recommendations, but lack a management environment.

With respect to UML, there was only few research conducted. Next to the above mentioned state machines, recommendations for UML modeling barely exceeded textual completion support. One of these systems was presented by Kuhn [Kuhn, 2010]. He focuses on recommending names for methods and other textual elements in UML. Again, this could be included in our environment.

Other textual supporting systems are usually found in IDEs. There are simple code completion systems and content assist systems, reducing spelling errors and increasing programming pace. Moreover, these IDEs were enhanced by real recommender ideas as done in Eclipse by Code Recommenders [Bruch et al., 2008]. This is a more clever code completion based on a knowledge base that includes rankings for code suggestions. Moreover, Code Conjurer is a reactive IDE recommender system providing potentially missing artifacts [Hummel et al., 2008]. It is a source code search engine based on Merobase [W.Janjic et al., 2013].

3 Model Recommender Framework

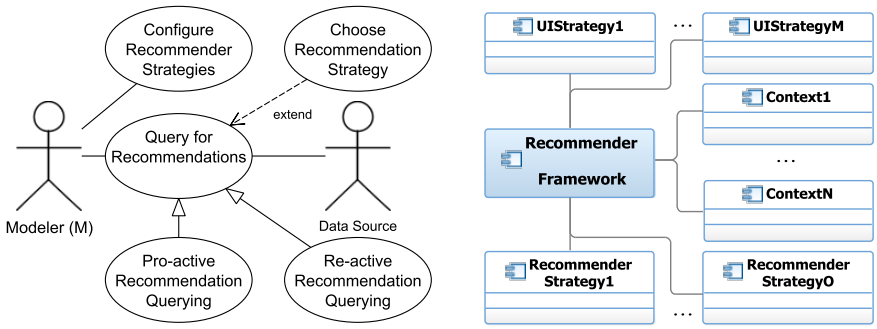
Since a framework should conform to requirements, we first briefly recite the needs and constraints. Then we quickly summarize the conceptual architecture that meets the requirements. After that, we implement this concept by an exemplary and more detailed realization. Unfortunately, we cannot explain each and every detail for the sake of brevity. An elaborated description is provided by Dyck [Dyck, 2012].

3.1 Requirements

Discussing possible requirements for a model recommender framework, we found the following functional and non-functional requirements. First of which are depicted in figure 1(a) as a use case diagram. Since we aim for framework support, there are only few externally visible requirements. Therefore, configuring the recommender strategies, querying for recommendations and choosing recommender strategies are all the functionality required.

Regarding non-functional requirements we found several necessities. First, multiple data sources or knowledge bases should be available for producing recommendations without the framework actually knowing about the concrete recommendation objects. In other words, the framework should be extendable. For

example, ontologies, ReMoDD, or MOOGLE should possibly serve as back-ends. Second, different algorithms should be pluggable into the framework, allowing multiple recommender strategies. Third, the context of the current editing should be captured, and thus, be taken into account for producing recommendations. This requires the same extendability as above, since a different editor might be regarded as another context. This leads to, fourth, the requirement to support several user interfaces because different recommendations might be presented differently in different editors. Fifth, the user interface should be non-blocking, i.e., responsive. This is important, as it might take a while until recommendations are produced. This leads to, sixth, decoupled and multi-threaded back ends. Last, the framework should be easy to use and provide support for starting extensions from scratch.



(a) Use Case Diagram (b) Architectural Concept

Fig. 1. Model Recommender Framework: Use Cases and Architecture [Dyck, 2012]

To sum up the requirements, a framework needs to allow for simple extension (cf. figure 1(b)). To the best of our knowledge we could not find such a research environment.

3.2 Framework Extensions by Example

The requirements described above lead to a conceptual architecture as depicted in figure 1(b). It shows a core which is extendable in three respects. First, it allows a `RecommenderStrategy` to be plugged into it. Therefore, data is gathered and processed in such strategies. Eventually, recommendation objects are produced in recommender strategies. Second, a `Context` builds the bridge between produced recommendations and an editor to have it applied to. This means a `Contexts` links these two as well as it adapts, if necessary. Third, a `UIStrategy` is a means to trigger queries and to depict results. The easiest example for a `UIStrategy` is a query box as depicted in the middle of figure 3. It works reactively because it needs to be opened explicitly. Another `UIStrategy` might be

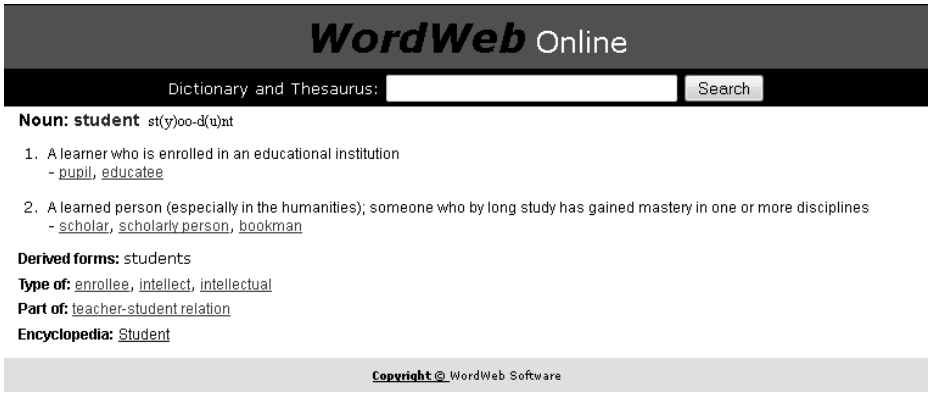


Fig. 2. WordWeb Online: Student

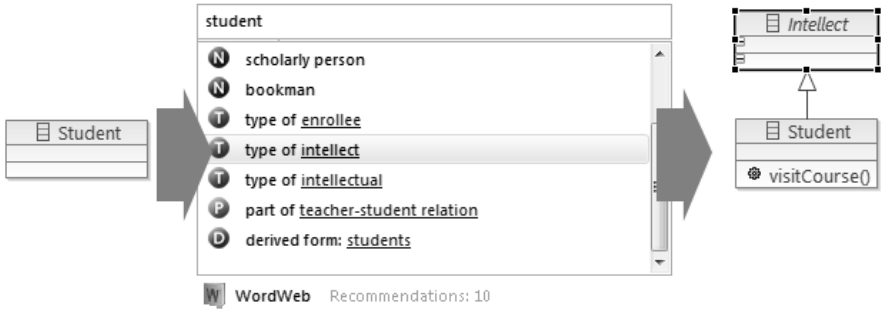


Fig. 3. Steps working with a Searchbox and a WordWeb RecommenderStrategy

a view that follows a cursor position and produces recommendations based on the next neighbor information related to the mouse position, i.e., it would be pro-active.

Subsequently, we explain how the framework needs to be extended by making use of each hot spot with an example as depicted in figure 4. Our example assumes a deployed framework operating on a class diagram canvas comprising a class `Student` as shown in figure 3 on the left. Next, a searchbox is opened by invoking `Ctrl+Space` waiting for a query. Then, after querying and applying a recommendation, the canvas looks like the right hand side of figure 3. Please mind that WordWeb Online is not a clever data source and that we do not pick the most obvious item gaining an abstract super class called "Intellect" due to a "type of" entry.

To begin with, the recommender strategy hot spots are `Recommendation` and `RecommenderSearchStrategy` (cf. figure 4). The first has to realize a method `apply()` that is invoked, if a `Recommendation` object is to be applied in an editor. The latter class has to realize the actual `search()`. In particular, if in-

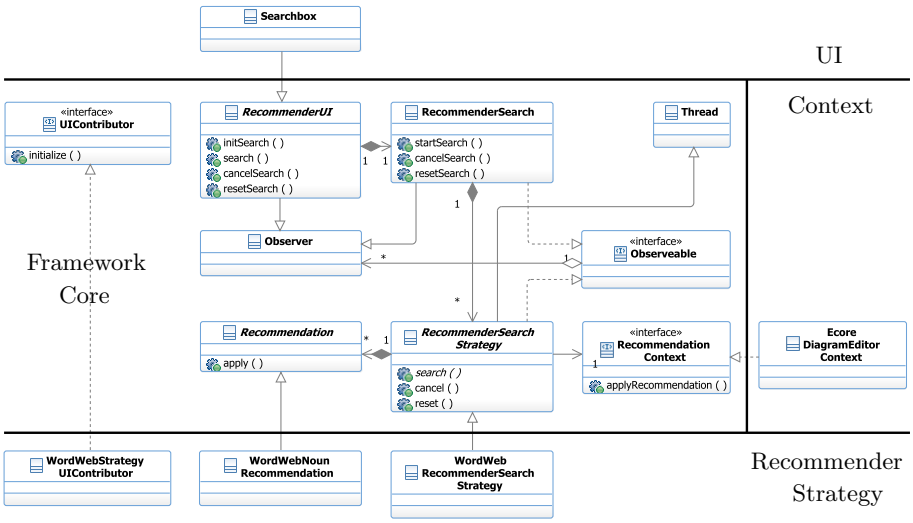


Fig. 4. Framework with Extensions: Searchbox, EcoreDiagramEditor, and WordWeb

voked on our `WordWebRecommenderSearchStrategy` object, this method is kind of the `main()` method to the strategy. In our example, WordWeb Online is queried with "student" and the found web page (cf. figure 2) is parsed leading to, e.g., `WordWebNounRecommendation`s (cf. figure 3 middle). Other kinds of `Recommendations` are possible as well, each of which with their own behavior, i.e., `apply()` method. This is why a recommender strategy can register labels different for each of its `Recommendations`; we can produce several different kinds and `WordWebNounRecommendation`s are among them. Mind that the framework knows about the labels because they are registered to it by the `WordWebStrategyUIContributor`.

The next hot spot is for user interfaces. In our example, the `Searchbox` defines how it should look like, how a `search()` is started (e.g., time delay), how a search is canceled (e.g., `Esc`), and how other operations turn out (e.g., `Ctrl+Space`). Since quite a lot of this is similar for several `RecommenderUI`s some default implementation is provided by the core through `RecommenderUI`.

Last, the context allows identifying editors and linking our `Searchbox`s to editors. Hence, our `EcoreDiagramEditorRecommendationContext` is responsible for this editor only. Moreover, this context knows how the `apply()` from our `WordWebNoundRecommendation` object has to be executed. In our example, EMF compound commands are created for the business objects and a drop action creates the visual representation on the canvas [Steinberg et al., 2009]. Please note that in figure 3 a super class was created since a `WordWebTypeOfRecommendation` was picked and the sub class, i.e., `Student` already existed on the canvas. This is due to the context realization and how it handles collisions with existing entities.

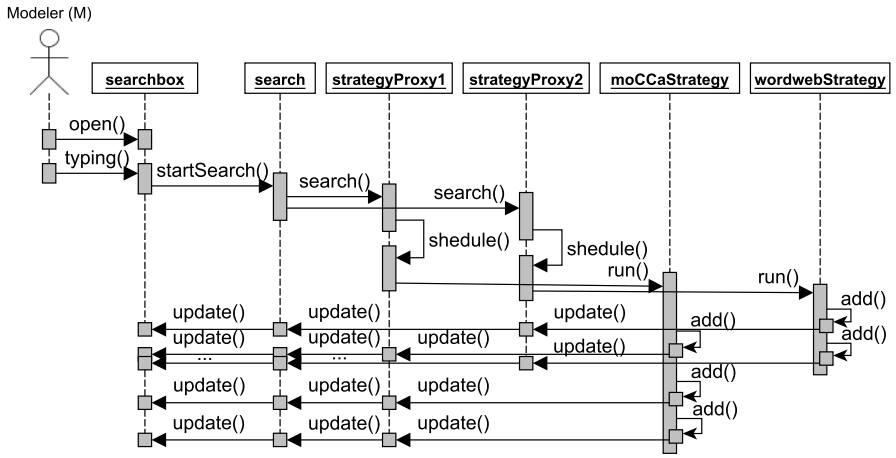


Fig. 5. Sequence Diagram: `search()`

3.3 Framework Object Flow by Example

So far this mostly gray-box view on the framework allows to get started with model recommendations. However, more details are necessary, if advanced features are desired. Therefore, we elaborate on internal factories, threading, notification mechanism, and proxies below. A very basic example in form of a sequence diagram is depicted in figure 5 leveraging our above example, i.e., the WordWeb strategy and the Searchbox, will complement the explanation.

First, as a UI triggers a search, the framework uses a search factory to instantiate each registered `RecommenderSearchStrategy`; in our example, this is the `WordWebRecommenderSearchStrategy`. Then the `RecommenderSearch` starts the actual strategies and feeds them with the query information. Finally, they return their `Recommendation`s as they are done. This is possible, because each `RecommenderSearchStrategy` is an `Observable` with observing `RecommenderUI`s. In our example, the `Seachbox` is registered as an `Observer` and can react on `update()` calls by our `WordWebRecommenderSearchStrategy`.

It was necessary to implement an observer pattern as explained above because `RecommenderSearchStrategies` are multi-threaded. We did so, because multiple searches should run simultaneously to enable quicker responses by strategies, e.g., one strategy might work locally already producing recommendations, while another one might work on a slow remote server still waiting for data. Moreover, every time a `RecommenderSearchStrategy` produced a `Recommendation` it should be possible to update the `UIStrategies`. That means, if our WordWeb server is very slow and each line from figure 2 is returned with a noticeable delay, a `Recommendation` object can be created for each line and sent to the `Searchbox` via an `update()`. In other words, the `Searchbox` would continuously

update showing more and more items. As a consequence, this required us to wrap `RecommenderSearchStrategy` objects in proxies.

In terms of a sequence diagram, the framework acts as depicted in figure 5. It shows how our realized UI, called `searchbox`, is opened and filled with some text. Then the actual `search()` is invoked, starting two proxies which encapsulate two realized `RecommenderSearchStrategies`. First, the `wordwebStrategy` is invoked. Second, a `moCCaStrategy` is invoked. This strategy queries an enhanced model library called MoCCa [Ganser and Lichter, 2013]. After that, both of the strategies work independently and `add()` their `Recommendation` objects as they produced them. Right after that they might call `update()` methods to delegate notifications to the `searchbox` which displays the found items.

If a `Recommendation` is picked, `apply()` is invoked. It finds out the current `RecommendationContext`, adapts the content of the `Recommendation`, and executes it. In our example from figure 3, a `WordWebNoundRecommendation` was picked, converted to an EMF compound command, and executed on the EcoreDiagram editor canvas, i.e., EditingDomain [Steinberg et al., 2009] [Eclipse, 2012].

An Eclipse P2 Updatesite and video of the running framework can be found on the web and on YouTube [Ganser, 2013b], [Ganser, 2013c].

4 Dashboard Support

Extending the framework explained above is easy, but requires manual tasks which can be easily done by tools. For example, extending classes or implementing interfaces by new classes is always the same. Moreover, registering a search strategy or a recommendation follows always the same patterns. This is why we ship a dashboard along the framework which offers user guidance and helps to jump start the framework in a few minutes. Figure 6 shows a screen shot of the dashboard [Schiller, 2013].

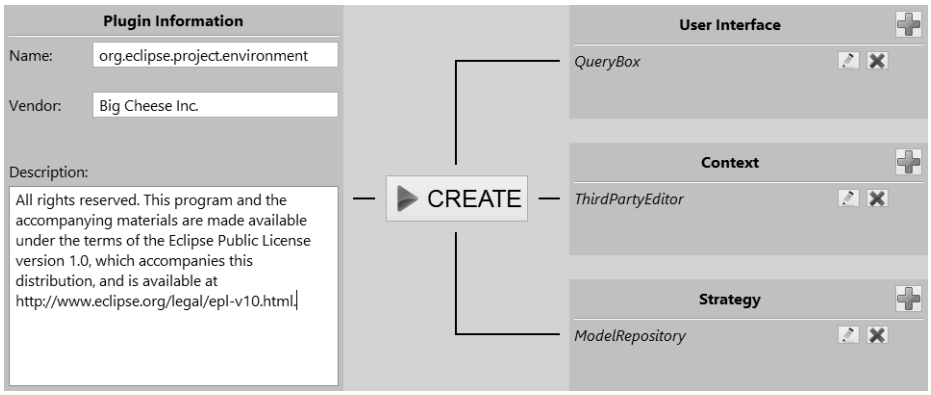


Fig. 6. Model Recommender Framework Dashboard

The left part of the dashboard is about general information, and the extensions part is on the right. The latter mirrors the extendable parts from the framework as shown in figure 1(b), enabling adding several UIs, as well as several contexts, and several recommender strategies. Moreover, certain configuration parameters are adjustable. For example, a strategy might use a directory, an FTP server, or an SQL data base as a data source. Hence, we included these default connectors in the configuration step.

Other than that, each entry on the right of the dashboard, will result in its own Eclipse plugin later on to adhere separation of concerns. Each will comprise ready to use classes and helpful skeleton source code including TODO comments. Moreover, it will contain all the configuration necessary to register this plugin and its hot spot extensions to the framework through Eclipse extension points.

5 Conclusion and Future Work

The field of model recommenders is rather new and will need a lot of research until high-quality recommendations can be produced as in other domains. Unfortunately, adjusting the known algorithm and applying them to models does not work. Thus, we created a research environment that is meant to enable experimenting with model recommender UIs and model recommender strategies, i.e. algorithm. This environment comprises a software framework as explained in section 3 and other tool support as explained in section 4. It was realized in the context of the HERMES project [Ganser, 2013a], it is available as an Eclipse P2 Updatesite [Ganser, 2013b], and a video shows its functionality, [Ganser, 2013c].

In more detail, we, first, explained a bit on the conceptual architecture of the actual software and how it can be extended. The point was that several UIs, contexts, and recommender strategies are required due to several possible deployment scenarios. Second, we elaborated on the details with an exemplary realization and illustrated the calls in a sequence diagram. Last, we described our framework dashboard, which is meant as initial user guidance.

Objectives of publication and future work are: First, tool support that comprises a simulation environment that eases developing recommender strategies. Second, an enhanced context management that provides contextual information to recommender strategies. Third, a template engine that allows for building place holder into models and offering user guidance while model templates are applied. And, last, the actual algorithm how to produce good recommendations based on enhanced model libraries like MoCCa [Ganser and Lichter, 2013].

Last, but most importantly, we hope to provide a useful and easy to use framework for model recommender research. And, we are very excited and curious about community feedback since each and every discussion we had on model reuse let to the consensus that there is huge need and potential.

Acknowledgments

We would like to thank all our reviewers for their comments! We would also like to thank Daniel Schiller and Viet Ngoc Tran for their contributions.

References

[Bruch et al., 2008] Bruch, M., Schäfer, T., and Mezini, M. (2008). On evaluating recommender systems for api usages. In *Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, RSSE '08, pages 16–20, New York, NY, USA. ACM.

[Dyck, 2012] Dyck, A. (2012). Recommender System Architecture for Ecore Libraries *(Master Thesis, RWTH Aachen University)*.

[Eclipse, 2012] Eclipse (2012). Ecore Tools. http://wiki.eclipse.org/index.php/Ecore_Tools.

[Ganser, 2013a] Ganser, A. (2013a). Reusing Domain Engineered Artifacts for Code Generation – The HERMES Project (Harvesting, Evolving, and Reusing Models Easily and Seamlessly). http://goo.gl/4LRdN.

[Ganser, 2013b] Ganser, A. (2013b). The HERMES Project - Eclipse P2 Updatesite: HERMES.reuse. http://goo.gl/ZGxIf.

[Ganser, 2013c] Ganser, A. (2013c). YouTube: Model Autocompletion Demo. http://goo.gl/fqwxl.

[Ganser and Lichter, 2013] Ganser, A. and Lichter, H. (2013). Engineering Model Recommender Foundations. In *MODELSWARD 2013, International Conference on Model-Driven Engineering and Software Development*.

[Hummel et al., 2008] Hummel, O., Janjic, W., and Atkinson, C. (2008). Code conjurer: Pulling reusable software out of thin air. *Software, IEEE*, 25(5):45–52.

[Kuhn, 2010] Kuhn, A. (2010). On recommending meaningful names in source and uml. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 50–51, New York, NY, USA. ACM.

[Maalej et al., 2013] Maalej, W., Robillard, M., Walker, R., and Zimmermann, T. (2013). Recommendation Systems for Software Engineering (RSSEs). http://goo.gl/zVqTK.

[Mazanek et al., 2008] Mazanek, S., Maier, S., and Minas, M. (2008). Auto-completion for diagram editors based on graph grammars. In *Visual Languages and Human-Centric Computing, 2008. VL/HCC 2008. IEEE Symposium on*, pages 242–245.

[Schiller, 2013] Schiller, D. (2013). Cockpit Support for Ecore Library Recommender *(Bachelor Thesis, RWTH Aachen University)*.

[Sen et al., 2008] Sen, S., Baudry, B., and Vangheluwe, H. (2008). Domain-specific model editors with model completion. In Giese, H., editor, *Models in Software Engineering*, volume 5002 of *Lecture Notes in Computer Science*, pages 259–270. Springer Berlin Heidelberg.

[Sprague, 1980] Sprague, R. H. (1980). A framework for the development of decisoin support systems. *MIS Q.*, 4(4):1–26.

[Steinberg et al., 2009] Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition.

[White and Schmidt, 2006] White, J. and Schmidt, D. C. (2006). Intelligence frameworks for assisting modelers in combinatorically challenging domains. In *In Proceedings of the Workshop on Generative Programming and Component Engineering for QoS Provisioning in Distributed Systems at the Fifth International Conference on Generative Programming and Component Engineering (GPCE)*, page 90.

[W.Janjic et al., 2013] W.Janjic, Hummel, O., Schumacher, M., and Atkinson, C. (2013). An unabridged source code dataset for research in software reuse.