A Framework for Model Recommenders
Requirements, Architecture and Tool Support

Andrej Dyck, Andreas Ganser, and Horst Lichter
Software Construction, RWTH Aachen University, Ahornstr. 55, 52074 Aachen, Germany
{*dyck, ganser, lichter*}*@swc.rwth-aachen.de*

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

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 present the requirements, concepts, architecture, and the realization below. Last of which is easily extendable and adaptable to either new data recommendation strategies or new environments like editors. As additional tool support, we provide a simulation environment, which ease development as well as implementing recommendation algorithm. Accordingly, researchers get all the conceptual groundwork and a realized infrastructure that ease the initial burden to start recommendations in modeling environments.

1 INTRODUCTION

Humans have always been hunter-gatherer seeking to put things away for later reuse. Generations collected pictures from, e.g., family events in order to skim through these pictures later, where order is essential for how efficient they can be found later. This is why generations over generations reinvented labeling systems to stay on top of things.

Considering this information management problem in the context of software artifacts, one notices that quite the same techniques were used for a long time. Documents were categorized, content was indexed, and search algorithms were invented. However, the information overflow could barely be managed. For example, a programmer needs to be familiar with many APIs, so often the overview is quickly lost. Due to that, type completion mechanism support programmers and help on a syntactic level. Yet, recent programming environments go further and provide good guesses of what might happen next (Eclipse, 2012a). The underlying ideas are taken from recommender systems, which proved beneficial for web shops by recommending products to customers (e.g., Netflix).

Another domain that could benefit from recommender system support is modeling. Note that at first glance the recommendations in modeling environments appear similar as for programming – but they are not. Considering UML class diagrams, firstly, this is due to the weaker semantics of the underlying concepts. While recommendations for Java source code need to compile, recommendations for class diagrams just need to be valid and well formed. Secondly, there are numerous kinds of data available: model repositories, glossaries, dictionaries, ontologies, and so on. They all allow for producing different recommendations, i.e., names, hierarchies, or partial models.

Investigating the limitations, which might be put on producing recommendations for class diagrams, it proved helpful to distinguish three areas. First, the content itself limits the kind of recommendations as mentioned above. Second, the current context influences the appropriateness of a recommendation. For example, 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 restricts how recommendations are presented since pro-active and re-active systems work differently (Dyck et al., 2014).

Altogether we researched how a recommender framework could support conducting research on model recommenders. Consequently we contribute a framework and a simulation environment, so model-

ers can jump-start model recommendations with their very own model recommender system. All that is needed is to implement the actual algorithm, i.e., most of the scaffolding is provided by the easily extendable framework. Therefore, we first explain the requirements we found for such a framework (section 3.1), explain its basics (section 3.2), elaborate on internals (section 3.3), and describe the simulation environment (section 4).

2 RELATED WORK

Recommender Systems could be discussed rooted in Information Management Systems and Decision Supporting Systems (DSS) in the eighties (Sprague, 1980). But terminology evolved, so we keep to the recent state and look into more recent frameworks without contrasting them to DSS.

Regarding UML, there was only few research conducted, so recommendations for UML modeling barely exceeded textual completion support. One of these approaches was described by Kuhn (Kuhn, 2010) focusing on recommending names for textual elements of the UML like methods. An investigation on architectural aspects was not addressed.

Another approach was presented by Sen et al.'s (Sen et al., 2008) which bases on Prolog. They demonstrate what they call "partial model completion" for finite state machines. Moreover, they offer a brief methodology. Our work differs w.r.t. target environments, because we try to keep it as wide as possible, meaning that we could use their solution and plug it into our framework by adapting their interfaces.

Looking at a broader scope, White and Schmidt present a framework for domain specific modeling languages on a conceptual level (White and Schmidt, 2006). But they focus on establishing domain specific knowledge bases and algorithms. They do so to be able to work in "combinatorically challenging domains". Hence, they use Prolog and demonstrated their approach by means of an AUTomotive Open System ARchitecture (AUTOSAR) example. We, in contrast, do not focus on domains or editors, but try to provide a conceptual and implemented infrastructure. One could use their implementation as a recommender strategy in our framework.

With Nassi-Schneiderman diagrams, triple graph grammars are used as a foundation by Mazanek et al. (Mazanek et al., 2008). They transform the diagrams into graph grammars, which they then leverage for auto-complete mechanism. In other words, they produce suggestions and these we could plug their realization into our framework as a strategy.

Other supporting systems are of textual nature and usually found in IDEs. Recent supporting systems exceed code completion systems and content assist systems by means of recommender system ideas. For example, an Eclipse project called Code Recommenders (Bruch et al., 2008) is such a system. It is a more clever code-completion-based on a ranking enhanced knowledge base for code suggestions. Another example is Code Conjurer, a reactive IDE-recommender system that provides potentially missing artifacts (Hummel et al., 2008) using a source code search engine called Merobase (Janjic et al., 2013).

3 THE FRAMEWORK

We firstly summarize the needs and constraints for a model recommender framework. Then we explain our conceptual solution and elaborate on our 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 Black-box Aspects: Requirements

Discussing possible requirements for a model recommender framework, we found the following functional requirements that are depicted in figure 1 as a use case diagram. Since we aim for framework support, there are only few externally visible actors and requirements. Therefore, only a modeler and a data source are involved for configuring the recommender strategies, querying for recommendations and choosing recommender strategies.

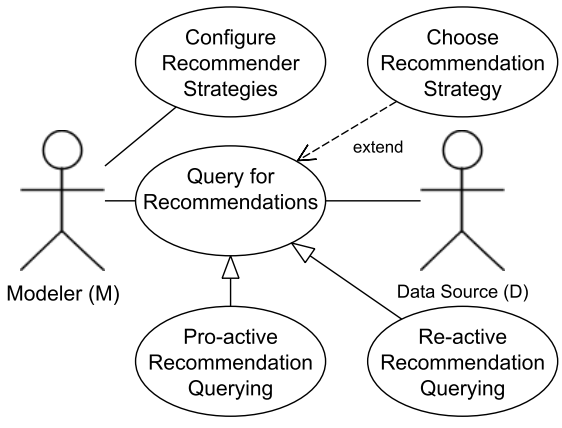

Figure 1: Model Recommender Framework Use Cases

Regarding non-functional requirements we found several necessities. First (1), multiple data sources

(e.g., repositories or knowledge bases) should be queried for producing recommendations, while the framework must not know about the concrete content of the recommendation objects. For example, ontologies, ReMoDD (France et al., 2007), MOOGLE (Lucrdio et al., 2010), or MoCCa (Ganser and Lichter, 2013) should possibly serve as back-ends. Second (2), different algorithms should be pluggable into the framework, allowing multiple recommender strategies. Third (3), the context of the current editing should be captured, and thus, be available for producing recommendations, e.g., querying, ranking or filtering. This requires the same extendability as above, since a different editor might be regarded as another context (Dyck et al., 2013). This leads to, fourth (4), the requirement to support several user interfaces, since different editors might present recommendations differently (Dyck et al., 2014). Fifth (5), 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 (6), decoupled and multi-threaded back ends. Last, the framework should be easy to use and provide support for starting extensions from scratch.

To sum up the requirements, the model recommender framework needs to realize a core that is extendable (cf. figure 2). To the best of our knowledge, there is no such environment available.

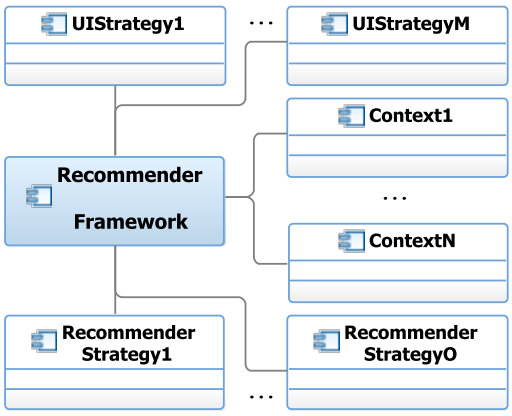

Figure 2: Conceptual Architecture

3.2 Gray-box Aspects: Hot Spots

The requirements described above lead to a conceptual architecture as depicted in figure 2. It shows a core which is extendable in three respects: First, it allows a `RecommenderStrategy` to be plugged into it (requirements (1) & (2)). Therefore, data is gathered and processed in such strategies and, eventually, recommendation objects are produced and handed over to the framework. Second, a `Context` builds the bridge between produced recommendations and an editor to have it applied to (requirements (3) & (4)). This means, a `Context` links these two as well as it adapts, if necessary. Third, a `UIStrategy` is a means to trigger queries and to depict results (requirement (4)). The easiest example for a `UIStrategy` is a query box as depicted in the top left corner of figure 8. It works re-actively because it needs to be opened explicitly as known from code completion. Another `UIStrategy` might be a view that follows a cursor position and produces recommendations based on the next neighbor information related to the mouse position, i.e., a pro-active system. Furthermore, supporting non-blocking UIs and multi-threading (requirements (5) & (6)), are properties of the framework. Subsequently, we explain how the framework needs to be extended by making use of each framework hot spot (c.f. (Pree, 1996)) with an example (cf. figure 3).

The most important hot spots are related to the classes: `Recommendation` and `RecommenderSearchStrategy`. The former has to realize a method `apply()`, which is invoked if a `ConcreteRecommendation` object is to be applied in an editor. In figure 8 that would be a pick of an entry in the list. The latter has to realize the actual `search()`, which, invoked on a `ConcreteRecommenderSearchStrategy`, starts this recommender strategy. Additionally, labels and icons related to `ConcreteRecommendations` can be registered by the `ConcreteUIContributor`. Due to that, a `ConcreteRecommenderSearchStrategy` can return several kinds of `ConcreteRecommendations` each represented by a different icon; e.g., the UI in figure 8 shows the label of our model repository.

The second most important hot spot regards the user interface, where a `ConcreteRecommendationUI` needs to realize how a `search()` is triggered and how the user can interact with the system, e.g., pick recommendations, cancel the search, etc. Since quite a lot of this is similar for several `ConcreteRecommenderUIs` some default implementation is provided by the core through the `RecommenderUI`. Finally, the `ConcreteRecommendationUI` needs to implement the graphical aspects as well (Dyck et al., 2014). For example, figure 8 shows a query box with a drop-down window as an overlay on a class diagram canvas.

Last, a new context is created by extending the hot spot `RecommendationContext`. By doing so, a `ConcreteRecommendationContext` identifies an editor and registers a `ConcreteRecommenderUIs` to

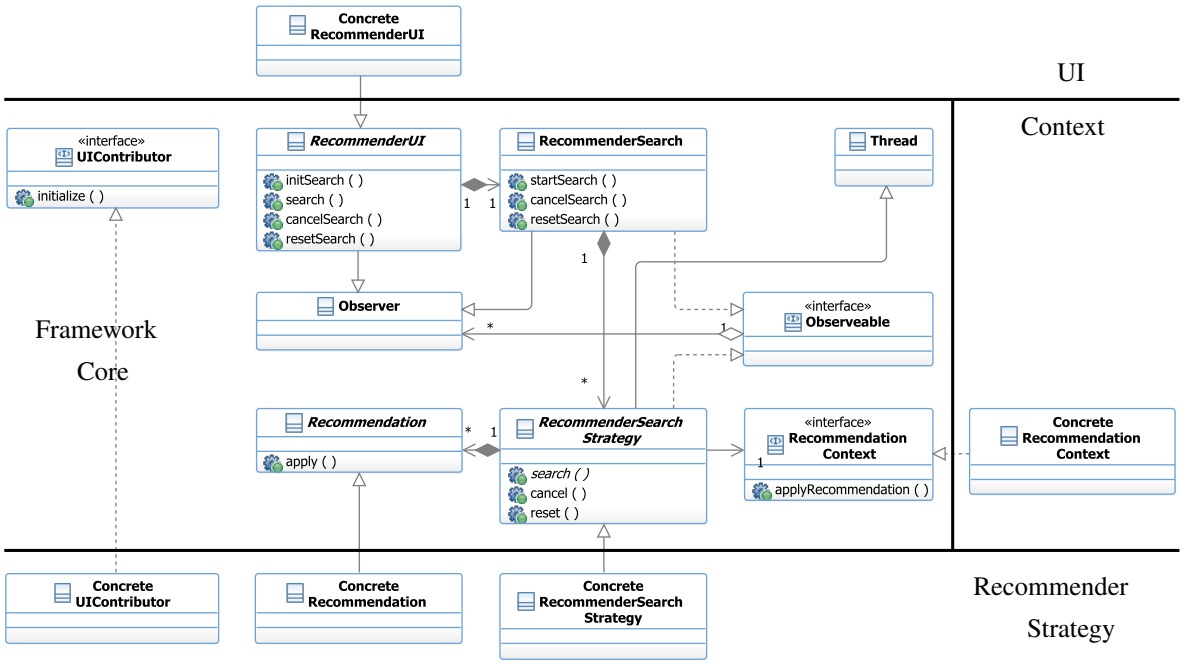

Figure 3: Coarse Grain Architecture

it. In other words, one context is mostly responsible for one editor. This is due to editors usually offering unequal sets of operations. This is why a context knows how the actual `apply()` from a `ConcreteRecommendation` object has to be executed, i.e., it adapts or wraps the operations if necessary (Dyck et al., 2013).

At first, all this looks fairly complex. However, we hope with our dashboard, which supports all of the extending and registering mentioned above, this complexity can be reduced. Anyhow, the details of the implementation comprise a few more technical details, since more sophisticated functionality is required to make the framework enjoyable for users.

3.3 White-box Aspects: Internals

So far a mostly gray-box view on the framework allows to get started with model recommendations, but more details are necessary for an advanced understanding. Therefore, we elaborate more on figure 3 and explain the factories, the notification mechanism, threading, the proxies, and the search strategy states. A very basic example in form of sequence diagrams will complement the explanation.

As a `RecommenderUI` triggers a `search()` method, the recommender framework uses a search factory that holds a set of all the registered `ConcreteRecommenderSearchStrategies` to instantiate an object each of them. Then the `RecommenderSearch` starts the actual strategies by invoking `startSearch()` and feeds them with the query information. This initiates all the instantiated `ConcreteRecommenderSearchStrategies` to run and waits for their results. Finally, they return their `Recommendations` and notify the `RecommenderUIs`.

Behind this notification mechanism is an observer pattern, which allows each `RecommenderSearchStrategy` to "tell" all the linked `RecommenderUIs` to update. This is possible, because each `RecommenderSearchStrategy` is an `Observable` and has `RecommenderUIs` as `Observers`. Mind that this allows continuous notifications, and therefore, for each recommendation in figure 8, to appear almost instantly. The benefit here becomes clear, when `Recommendation` objects are produced with a noticeable delay, e.g., due to a slow Internet connection. However, some recommender algorithm might not benefit from this feature because, e.g., collaborative filtering work on complete sets.

Note that, above, we started all the search strategies at once. That is possible, because we build in a threading mechanism that runs each `RecommenderSearchStrategy` object in its own thread, fulfilling requirements (5) & (6), i.e., have a non-blocking UI and have search strategies better load balanced. Again, considering a search strategy, which is delayed by a slow Internet connection, allows other strategies to proceed.

Yet another trick is necessary to avoid trouble

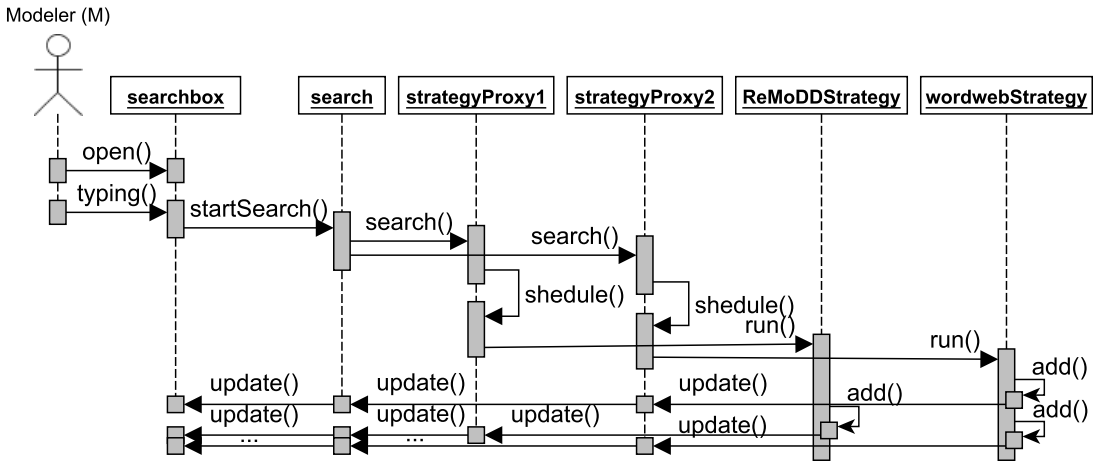

Figure 4: Sequence Diagram: `search()`

with search strategies. For the same reasons mentioned above, it makes perfect sense to wrap each `ConcreteRecommenderSearchStrategy` in a proxy. It binds the search strategy to the framework and acts as an adapter for the notification mechanism at the same time. For simplicity sake, we do not elaborate on this part of the architecture.

Finally, the proxies and threading decouple (see requirement (5)) the components to an extent that we had to realize a controlling mechanism for the recommender search strategies. To this end, we implemented a state mechanism as depicted in figure 5. It illustrates the life-cycle of a recommender strategy on two levels. First, on the outer level, a search strategy can be `enabled` or `disabled` – or `defect`. While the first two states can be set in the preferences, a `defect` strategy could be, for example, the result of a missing default constructor. Additionally, a search strategy can be asked to `reset`. This happens, e.g., if the UI is closed. Second, the internal level of a search strategy can follow a regular sequence of states, namely: `ready`, `running`, and `done`. Alternatively, it can be `failed` or `canceled`; where the former is due to an internal error the latter is being triggered by the user.

3.4 Sequential Aspects: Object Flow

For a better understanding of the framework's internal process, we provide an example by two sequence diagrams, shown in figure 4 and figure 6.

The first sequence diagram in figure 4 shows how a search starts and how it ends, delivering the recommendation objects to the UI. Please mind that we take some obvious shortcuts, while explaining the object flows compared to figure 3: First, a `Modeler` opens the `searchbox` (cf. figure 8) and starts typing. After a neglectable delay, the actual search is started, instantiating `strategyProxies` for each recommender search strategy. Since we implemented the framework for multi-threading, the proxies use a scheduling to queue up the actual search, which is eventually run. Doing so, the recommender search strategies are working independently, using their algorithm to produce recommendations. They can return these to the framework by storing them in their recommendations set and notifying the framework, which forwards the notifications all the way up to the registered observers, i.e., our `searchbox`. An example of the searchbox with a list of produced recommendations, provided by our recommender strategy, is shown in figure 8.

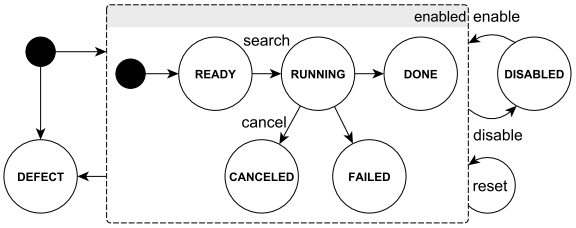

Figure 5: Search States

As the recommendations are listed in the UI, one of them can be picked as illustrated in the second sequence diagram in figure 6. It illustrates the steps how a recommendation is selected until it is applied to an exemplary editor. Again, we omit some obvious delegations to make the sequence diagram more comprehensible: First, the `Modeler` picks a recommendation in the `searchbox`. That invokes an `apply()` on a recommendation object, which needs to do some pre-

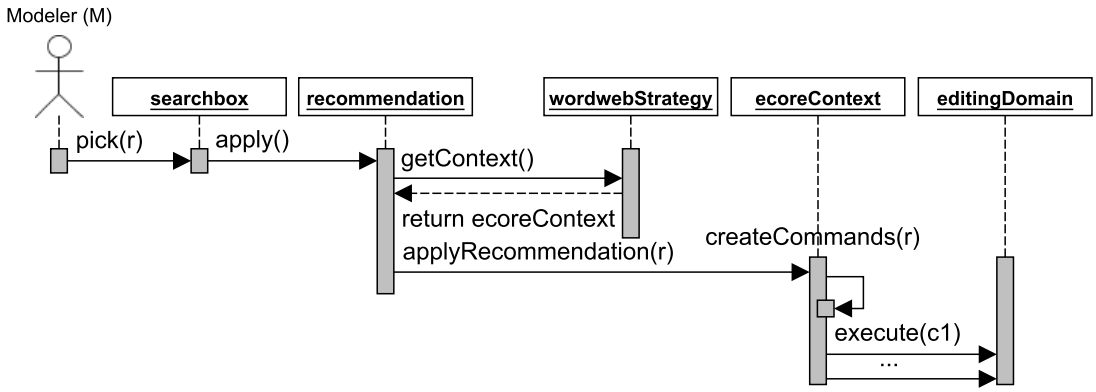

Figure 6: Sequence Diagram: `pick()`

processing. This is acquiring the context by asking the recommendation-providing search strategy about it. In this case, the recommendation was an entry on the WordWeb Online website (Dyck et al., 2013), which was parsed to several recommendation objects. Since the strategy knows about its context it returns an `ecoreContext` in our example. Now, the `apply()` can be started on the retrieved context. Hence, the recommendation object is taken and transformed into something that is applicable in an `ecoreContext`. This means, because our editor is able to execute EMF commands, a compound command is created by invoking `createCommands(r)` that are executed on the `editingDomain` eventually. Note that the EMF/Ecore-specific editing domain (Steinberg et al., 2009) is hidden behind the context object.

4 SIMULATION SUPPORT

So far, our model recommender framework provides a flexible, extensible, yet simple management environment. This is due to the few hot spots that are needed for development. In order to ease development and testing of recommender strategies, we implemented a simulation environment. It supports the developer finding malformed recommendation objects and errors while applying them in the context. Moreover, recommendation strategies, if build properly, can contribute in their very own way to parameter tweaking as we explain below.

4.1 Simulation Cornerstones

Speaking of simulation, there is a large theory behind how to design an appropriate simulation that actually meets the required needs and provides beneficial results (Banks, 1998). Based on this theory, we need to define what our object under simulation is, what we consider as our simulation environment, and what our simulation model is. Further, we need to define our simulation protocol. Since this is tightly connected to the simulation concept, it is reasonable to have glimpse at figure 7 to begin with.

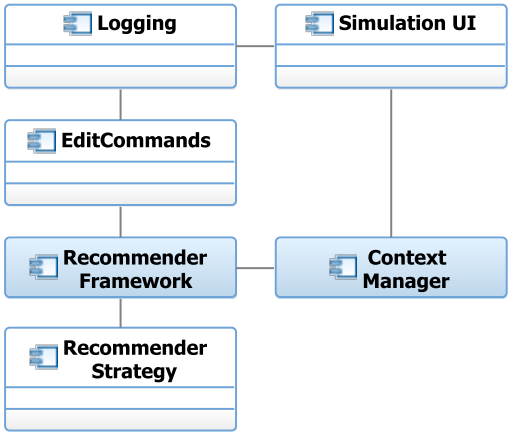

Figure 7: Simulation Architecture

First, our objects under simulation are recommendations and how they were produced by a recommender search strategy. In general, these recommendations are the results of a transformation of some source format to a command set which is applicable in graphical editors. Hence, monitoring these commands is the most promising approach to find out if a recommendation strategy produces the right recommendations. This makes the simulation a white-box simulation w.r.t. the recommendations and a black-box simulation w.r.t. the recommendation strategies; i.e., disregarding the latter system state variables.

The latter can easily be altered to become a white-box simulation if the programmer of a recommender search strategy adheres to certain rules regarding logging. Then, our simulation environment is able to trace the actual production of the recommendation as well as the recommendation algorithm, which is applied to the data back end, i.e., the use of candidate selection, filtering, and post processing. In other words, a white-box simulation of recommender strategies would be a monitoring of system state variables in terms of simulation theory (Banks, 1998). Often this is out of reach, so we discuss no more than the black-box properties first.

Second, our simulation environment is a graphical editor that is modifiable by a flexible command framework. This means, a recommendation, which is a set of commands, produced by a recommender search strategy, can be applied in this environment. This makes the whole environment a discrete event simulation and behaves very similar to GMF generated editors. They offer editing function through the EMF command framework, of which we make use of as well. However our edit command framework needs to be slightly more flexible, as we need to be able to react on unknown or malformed commands as well.

Altogether, this environment is stable, yet alterable serving as a controllable environment for simulations. It is stable, because it is stateless, always producing the same results for the same sequences of commands. Moreover, it is alterable, since the edit commands can be altered and extended by new commands to serve new kinds of recommendations.

Third, the simulation model is the foundation for our environment and needs to be the smallest possible basis of UML – or EMF models (Steinberg et al., 2009). This makes MOF – or Ecore – the obvious choice for the simulation model because they form the meta foundation in terms of grammars respectively. In other words, if this meta language is used, every model defined by it can be dealt with. Hence, this basis explains why there is a good foundation for the command framework and why it is as flexible as mentioned above. Additionally, these command frameworks would be quickly exchangeable if another simulation environment is needed.

Last, the simulation protocols are textual representations of attempts to apply recommendations in our simulation environment. This categorizes our approach as an activity scanning simulation (Banks, 1998). This is possible, since we have full control over our environment which would be almost impossible with third party environments. This means, all attempts and results are logged and can be analyzed.

We avoided to use a DSL as a foundation for the logging since we wanted it to be usable as easy as possible. In case we had used a DSL, we would certainly be able to analyze the simulation traces more quickly, but we would put this burden on users as well, i.e., learning the grammar or even parts of, e.g., xtext.

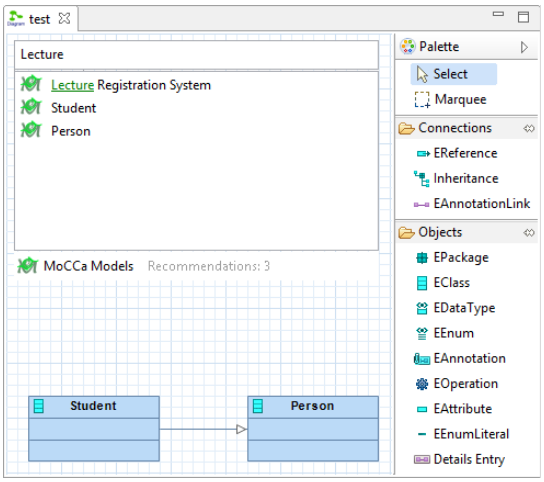

Figure 8: Simulation UI

4.2 Simulation Aspects and Concept

As we have set the foundations of our simulation, we can look into the details of the concept; how they fit together and what issues could be addressed. We can subdivide this along figure 7. First, the simulation UI, which is a replacement for the real editor to be used later, we can investigate on issues related to collaboration between recommendations and editors. Second, the recommendation's consistency is under investigation as well, i.e., if their application results in valid models. And, last, the recommendation objects need to be produced, so the recommender strategy is under investigation as well.

It can be rather tedious to find out the reasons why a recommendation results in a rather unexpected editor canvas. The reasons can be manifold, but the best approach to find out is using something similar to a test spy (Meszaros, 2006). In our case the recommendation should be applied to a canvas; thus, this should be the test spy. This means the canvas is replaced with another canvas that is able to record and log all the actions which are "executed" on it. This means, for every known command an info message is created, while malformed commands result in warnings and unknown commands result in error messages. While known and unknown commands are rather self-explanatory, an example for a malformed command is one that tries to create a method with parameter type that is not in the current scope.

All in all our simulation user interface is a rewritten class editor by means of the Graphiti framework (Eclipse, 2012c). It is enhanced with logging functionality as mentioned above and serves as a controlled yet customizable environment. As the commands mirror the real third party editor, successfully applied recommendations should work in the final environment as well.

Certainly this requires valid recommendation objects, but these can be invalid in three ways. Therefore, each recommendation object needs to be checked (1) if it produces well formed models. This can be done by Ecore Tools or other available frameworks (Eclipse, 2012b). Moreover, (2) the recommendation can be incompatible with the simulation environment. This would be the case, if a UML command set was created instead of an Ecore command set. Last, (3) the recommendation could hold commands that are not denotable by the simulation environment. While these errors could be logged as unknown commands, the semantic is slightly different. For example, a command set might contain stereotype information, while the simulation environment is not able to "understand" these, a whole set of information is not applicable. The reason for this could either be a wrong filtering by the recommendation strategy or a wrong transformation of the source data to commands. However, both of these issues might be rooted in a misinterpreted query or insert context.

Regarding how recommendation objects are produced, some aspects can be monitored, while others maybe not. For example, a recommendation strategy has some internals on how to rank and transform objects which might not be exposed. We can investigate in- and outgoing data and refer to them as the query context and the insert context. From them and the provided data source, we can gain useful information, if a recommendation strategy works appropriately. For example, an editing position, which is the text field of a class name, should not get a recommendation that wants to create a whole class in this text field. In this case, we could assume that the post filtering in the recommendation strategy is not correct and that it probably did not take into account the query context.

As we make appropriate use of logging in a recommender strategy, we changed to a white-box simulation. There we can gain insights to the algorithm and use the system state variables to find errors or tweak the algorithm. This is possible if the dashboard was used to generate the recommender strategy and the logging is used appropriately for parameters and calculation details. Because only then, we are able to see how the query and query context lead to source data, how this source data is ranked, filtered, and transformed into recommendation objects.

The benefit of using this approach appears to be that no debugging is necessary and no information noise as known from debugging environments is distracting developers. Hence, the developer can focus on evaluating the parameters, monitoring the ranking, and analyzing the recommendation objects. In addition, tools like Logback-beagle (Glc et al., 2012) enable easy gathering and evaluating of the logging, e.g., "jump to log position", and thus, allowing easy tweaking of a recommendation algorithm.

5 CONCLUSION

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 algorithms 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 search strategies, i.e., algorithms. This environment comprises a software framework as explained in section 3 and tool support in form of an simulation environment 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 search 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 simulation environment, which is meant as developer support for developing and testing recommender search strategies.

In this paper, due to lack of space, we omitted to mention a further tool support; a dashboard that offers user guidance and helps to jump-start the framework. This dashboard guides a user through important configuration steps and creates plug-ins for the hot spots described in section 3.2. The former comprise ready to use classes and helpful skeleton source code, along with all the necessary configurations.

Objectives of publication and future work are: First, an enhanced context management that provides more detailed contextual information to recommender search strategies. Second, a template engine that allows for building place holder into models and offering user guidance while model templates are applied.

And, last, concepts and algorithms 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. 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.

ACKNOWLEDGEMENTS

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

REFERENCES

Banks, J. (1998). *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice*. A Wiley-Interscience publication. Wiley.

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

Dyck, A., Ganser, A., and Lichter, H. (2013). Enabling Model Recommenders for Command-Enabled Editors. In *MoDELS MDEBE - International Workshop on Model-driven Engineering By Example 2013 co-located with MoDELS Conference, September 29, 2013, Miami, Florida*.

Dyck, A., Ganser, A., and Lichter, H. (2014). On Designing Recommenders for Graphical Domain Modeling Environments. In *Modelsward 2014, Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development, Lisbon, Portugal, 7.-9. January 2014*. SCITEPRESS.

Eclipse (2012a). Code Recommenders. http://www.eclipse.org/recommenders/.

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

Eclipse (2012c). Graphiti. http://www.eclipse.org/graphiti/.

France, R., Bieman, J., and Cheng, B. (2007). Repository for model driven development (ReMoDD). In *Models in Software Engineering*, volume 4364 of *Lecture Notes in Computer Science*, pages 311–317.

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

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

Ganser, A. and Lichter, H. (2013). Engineering Model Recommender Foundations - From Class Completion to Model Recommendations. In *Modelsward 2013, Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development, Barcelona, Spain,19.-21- February 2013*, pages 135–142.

Glc, C., Pennec, S., and Harris, C. (2012). Logback-beagle. http://logback.qos.ch/beagle/.

Hummel, O., Janjic, W., and Atkinson, C. (2008). Code Conjurer: Pulling Reusable Software out of Thin Air. *Software, IEEE*, 25(5):45–52.

Janjic, W., Hummel, O., Schumacher, M., and Atkinson, C. (2013). An Unabridged Source Code Dataset for Research in Software Reuse. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 339–342, Piscataway, NJ, USA. IEEE Press.

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.

Lucrdio, D., de M. Fortes, R., and Whittle, J. (2010). MOOGLE: A Metamodel-based Model Search Engine. *Software and Systems Modeling*, 11:183–208.

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

Meszaros, G. (2006). *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR.

Pree, W. (1996). *Framework Patterns*. SIGS Books and Multimedia, New York.

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, R. H. (1980). A Framework for the Development of Decision Support Systems. *MIS Q.*, 4(4):1–26.

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

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.