On Designing Recommenders for Graphical Domain Modeling Environments

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: Model, Recommender, Recommender Systems, User Interface and Interaction, Modeling Support, Survey

Abstract: Recommender systems for source code artifacts are newly emerging and are now successfully supporting programmers. Their underlying knowledge bases, recommender algorithms, and user interfaces are well studied. Integrated into the development environment, they do a fairly good job in reducing complexity and development time.
In contrast, research in recommender systems for domain modeling is widely missing. As a matter of fact, knowledge bases, studied as model libraries, are only a possible foundation but concerning recommender algorithms and user interface design research needs to be conducted.
Hence, we provide some foundations for graphical user interface design by answering how domain model recommender systems should look and feel like in graphical environments. To do so, we conducted a three-phased survey relating to modeling of UML class diagrams. Most importantly, we found that various user interfaces are required to meet different user needs. Finally, several algorithms are desired for diverse knowledge bases and diagram types; hence, leading to a demand for a flexible recommender architecture.

1 INFORMATION OVERFLOW

In order to make use of masses of information, several approaches emerged. Most popular among these are certainly recommender systems (Resnick and Varian, 1997). These became famous through Amazon (Linden et al., 2003) and Netflix (Bennett and Lanning, 2007), which recommend related products alongside search results. For example, a query for a book or video also presents products that were bought or viewed by customers with similar liking. In doing so, product portfolios become much more accessible for customers and they are not drowning in information overflow any more. In other words, recommender systems help users to overcome the information overload problem by exposing users to the most interesting items, and by offering novelty, surprise, and relevance (Systems, 2009).

Looking into software development, the amount of artifacts is steadily increasing as well and managing these became a major issue in lots of development projects. One kind of these artifacts is source code. Hence, state of the art integrated development environments (IDEs) offer query mechanism and code completion functionality to ease information overflow. Furthermore, ideas from recommender systems were recently adapted to bolster code reuse. For example, an Eclipse project called Code Recommenders learns from existing code bases to provide best guesses of what the programmer might want to do next (Bruch, 2012), (Bruch, 2008), (Weimer et al., 2009), (Eclipse, 2012). This means, as a programmer just created a new text object (cf. figure 3) the code recommender will offer a set of methods which other programmers invoked on such objects.

Other artifacts, which are often designed in software development, are domain models. Unfortunately, there is no recommendation support for modelers available yet. This is quite surprising, since a lot of effort has been put in researching model libraries and how to preserve modeling experience. Hence, the data which could form the knowledge base for a recommender is available, but, to the best of our knowledge, no efforts have been undertaken to make it reasonably accessible. This means, recommender algorithms and user interfaces are unavailable.

In fact, modeling tools are lacking in usability and barely support modelers beyond model validation and automatically arranging elements (Bobkowska and Reszke, 2005). As a consequence, modelers need to integrate models from knowledge bases manually from such sources if found at all (cf. figure 1). This

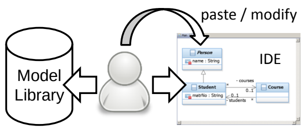


Figure 1: Model Reuse: How it is

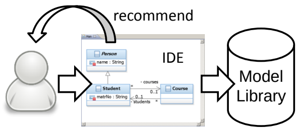


Figure 2: Model Reuse: How it should be

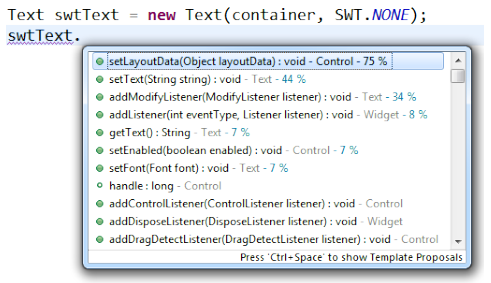
means, each time modelers want to reuse models from a library, they are distracted from modeling because they need to mediate between the modeling tool and the model library. Obviously, this has most likely a negative impact on the time spent and the quality achieved in modeling; therefore, making model reuse unappealing.

We want to put forward a design for an integrated recommender as illustrated in figure 2. Here, the developer interacts with the modeling tool only, which uses model libraries and recommender algorithms to recommend models for reuse. Unfortunately, there is no common sense on how a user interface should look and feel like. This is why we conducted a three-phased survey to find out how a graphical recommender should look and feel like for domain modeling. First, we study existing approaches from related domains in section 2. Second, we define the survey in section 3 explaining our methodology in section 3.1 as a three-phased approach. After that, we analyze the results in section 4 and discuss further ideas in section 4.4. Finally, we conclude with highlighting the most important findings and discuss future work in section 5.

2 RELATED RECOMMENDERS

Until lately, most recommender systems have been mostly tied to the web, e.g., as part of commercial systems, such as Amazon's recommenders. However, recommender systems specific to software engineering are emerging to assist software developers in a wide range of activities including code reuse (Robillard et al., 2010). Most of these systems are integrated in an IDE and suggest software artifacts, such as code snippets, and focus on *"you might like what similar developers like"* scenarios. A detailed discussion is provided by Happel and Maalej in (Happel and Maalej, 2008).

Three code recommender systems have inspired us. First, the project *Code Recommenders* is a recommender system for the Java programming language and is integrated into the Eclipse IDE (Bruch, 2012), (Bruch, 2008), (Weimer et al., 2009), (Eclipse, 2012). It comprises various intelligent code completion engines and documentation providers. For example, its intelligent call completion recommends only methods which will most likely be called at the current editing position (cf. figure 3).



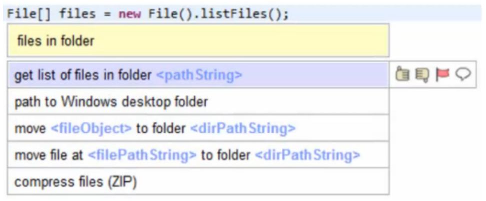
```
Text swtText = new Text(container, SWT.NONE);
swtText.
```

```
● setLayoutData(Object layoutData) : void - Control - 75 %
● setText(String string) : void - Text - 44 %
● addModifyListener(ModifyListener listener) : void - Text - 34 %
● addListener(int eventType, Listener listener) : void - Widget - 8 %
● getText() : String - Text - 7 %
● setEnabled(boolean enabled) : void - Control - 7 %
● setFont(Font font) : void - Text - 7 %
○ handle : long - Control
● addControlListener(ControlListener listener) : void - Control
● addDisposeListener(DisposeListener listener) : void - Widget
● addDragDetectListener(DragDetectListener listener) : void - Control
                          Press 'Ctrl+Space' to show Template Proposals
```

Figure 3: Code Recommenders' Call Completion (Eclipse, 2012)

Code Recommenders' dynamic templates completion take this to the next level by recommending a complete sequence of method calls. To this end, it uses available open-source code repositories analyzing common code structures. These code templates can serve as additional documentation that quickly shows how an API can be used, and thus, save developers' time with APIs they are not familiar with.

Second, *SnipMatch* recommends common code snippets similar to Code Recommenders' dynamic templates. However, here the developer queries the system describing the task they want to accomplish (cf. figure 4) (SnipMatch, 2012). SnipMatch is now



```
File[] files = new File().listFiles();
```

```
files in folder
get list of files in folder <pathString>
path to Windows desktop folder
move <fileObject> to folder <dirPathString>
move file at <filePathString> to folder <dirPathString>
compress files (ZIP)
```

Figure 4: Querying SnipMatch (SnipMatch, 2012)

part of the Code Recommenders project.

Last but not least, *Code Conjurer* is a recommender system that uses code-search engines to deliver high-relevance software-reuse recommendations with minimal disturbance to a developer's workflow (Hummel et al., 2008). It seamlessly integrates code search and reuse functionality into the Eclipse Java development environment, and thereby, allowing the developer to search for reusable code, for example, by defining unit tests (test-driven search).

Code Conjurer delivers code recommendations as results which satisfy the tests and even generate adapter classes to match the interface specified in the tests, if needed.

3 SURVEY

With model libraries and autocomplete functions of IDEs in mind, we had a vision of bringing these two together to gain easy access from graphical modeling environments. To the best of our knowledge, existing autocomplete functions in modeling tools have barely exceeded completion of textual elements. In fact, an autocomplete function gives proposals for all possible syntactical tokens within a certain syntactical position. However, the graphical syntax of models, e.g., class diagrams, is fairly simple. Thus, instead of an autocomplete function, we now considered recommender systems, which then could recommend restructuring of a model w.r.t. the semantics of that model. In the interests of simplicity, we have restricted ourselves in this survey to class diagrams and tools to design such.

There already exist recommender systems embedded into IDEs focusing on code reuse. However, if one considers modeling as a graphical task, one could imagine that the model recommender UI has to differ from UI of textual recommenders. Moreover, the recommender system can be triggered, either re-actively by the user, or pro-actively by the system. As a consequence, we have to consider different UIs for both cases. In this survey, we studied possible interactions with different UIs for recommending models and asked users what recommendations for class diagrams could be. As a constraint, we assumed that an ideal interaction is embedded into the modeling environment.

3.1 Methodology

The conducted survey consists of three phases in which we determined possible UIs for a model recommender. Starting with brainstorming and sketching mock-ups, we continued with guided interviews, and finally, we collected further opinions with an online questionnaire.

In the first phase, we did several sessions of brainstorming, where we gathered ideas for a model recommender system and sketched mock-ups showing possible graphical user interfaces (GUIs). We used the mock-ups to show users what a GUI might look like.

In the next phase, we conducted guided interviews with several Ph.D. students in order to refine and improve the designs. Since all of the participants have an extended knowledge in modeling and user interface design, we included a brainstorming session at the end of these interviews as well. Here, the participants should consider themselves using a build tool and wonder if and how it would blend in and support their modeling. This unveiled further design and interaction ideas, which we discussed iteratively with other Ph.D. students.

In the last phase, we condensed the gathered ideas and put them into an online questionnaire. The overall goal of this was twofold. First, we wanted to get participants opinion on how beneficial each user interface would be, and second, we wanted to stimulate further thinking on the topic. Hence, thoughts on this topic could be noted at the end of the questionnaire. The participants in this phase can be subdivided into three groups. Firstly, related team members from the research project were asked to fill out the questionnaire. Secondly, computer science students in general were asked to participate. Lastly, we posted a link on a modeling board asking members to contribute (EMFBoard, 2012).

3.2 Conducted Survey

As prototyping is an important technique to reduce risks by identifying design flaws before implementing the actual system, we have drawn mock-ups upfront and in multiple steps. While iterating over model recommender UI sketches in the brainstorming sessions, we tried to answer the following questions: *How should the system be triggered best? What should the GUI look like? How could recommendations be presented, previewed, and picked?* In order to focus, we restricted our survey to class diagrams as models and assumed the modeling software is used on a PC with a pointing device, e.g., computer mouse, and keyboard. Our ideas of a model recommender and its GUI evolved with each session of brainstorming.

After having collected several ideas and sketched GUI mock-ups in brainstorming sessions, we designed a guided interview, which was conducted with five Ph.D. students at the department at our university. Two of the interviewees assessed themselves as modeling experts, two as advanced modeler, and one as a beginner. All interviewees have experience with today's modeling software like AgroUML, AndroMDA, EMF, IBM's Rational Architect, Yatta's UMLLab, Visual Paradigm, and other tools.

We started an interview by introducing recommender systems like Amazon's "Customers Who

Bought This Item Also Bought"-recommendations and Code Recommenders for programming Java with Eclipse as explained above. Then, we asked the interviewees what they think would be useful to be recommended while modeling and if they miss a recommender system in today's modeling software. Moreover, we asked them how they could picture the user interaction with a possible recommender system for modeling software, e.g., how the system should be triggered, or how the results and previews should be displayed. Finally, we showed our mock-ups to encourage further discussion. An interview lasted about 60 minutes on average.

Having all the results in mind, we designed a questionnaire, which was published online eventually. The questionnaire was subdivided into four sections: The first section covered general questions about modeling with software, about a possible recommender system in modeling tools, about EMF, and questions regarding our mock-ups. Therefore, after introducing the recommender system as an "intelligent autocomplete function" for class diagram modeling tools, we asked for the modeling expertise and if they used software tools for modeling. Moreover, we wanted to know if existing models are reused and where the users get these models from. The second section covered the question if participants ever missed a recommender system in today's modeling software. Furthermore, we were interested in what recommendations participants consider to be useful. In the third section the "Eclipse Modeling Framework" (EMF) arouse from curiosity how popular this framework is and how pleasant the work with it's editors is. It will not take any role in our results since it was not the focus of this survey. Still, these information can be used to study correlations in more details which is out of focus here. In the last section, we presented our mock-ups in a slide show and examined how appealing the mock-ups appeared. Moreover, we questioned what keyboard shortcuts participants expect and what kind of preview would fit best for small and big diagrams. Finally, we gave participants the possibility to add additional thoughts.

A participation in this survey took eight and a half minutes to fill in on average (standard deviation: five minutes). The survey was open for three weeks.

4 SURVEY RESULTS

As this survey was subdivided into three phases, we dedicate a results section to each; the mock-ups phase, the guided-interviews phase and the online-questionnaire phase.

4.1 Mock-ups

Initially, we experimented with sketchy mock-ups to do reasoning on how a recommender system for modeling tools might look and feel like. We explain this reasoning because we were following the questions mentioned subsequently:

1. How is the recommender system triggered? In general, there are two ways to trigger a recommender. First, *re-active* triggering, meaning information is provided after an explicit user request. This is comparable to source code autocomplete which is triggered by keyboard shortcuts in general. Second, with *pro-active* triggering the system identifies certain situations based on the users context and give recommendations automatically in a continuous, non-disruptive way. Happel et al. state that pro-active triggers are important to be considered for recommender systems and that current recommender systems do rarely follow this concept (Happel and Maalej, 2008). For example, such pro-active recommender systems exist in other areas; e.g., recommendations in e-shops like Amazon (Schafer et al., 1999) (Linden et al., 2003), context-aware traveler agents (Felfernig et al., 2007) (Al Tair et al., 2012), or recommendations while writing documents, e.g., manuals (Puerta Melguizo et al., 2007). Mind that the decision whether to use re-active or pro-active triggers has a strong impact on the GUI.

2. What does the GUI look like? The GUI is heavily influenced by the kind of trigger. Thus, we created GUI mock-ups accordingly. First, for a re-active recommender system we were inspired by autocomplete functions from IDEs. Since a modeling canvas in general is a graphical editor, we designed figure 5 to what we call a *Searchbox* similar to the GUI of the Eclipse plug-in SnipMatch (SnipMatch, 2012). The Searchbox is activated by a keyboard shortcut and provides a textfield, where the user can enter a query. The query together with the context, i.e., the current model, define the context for recommendations.

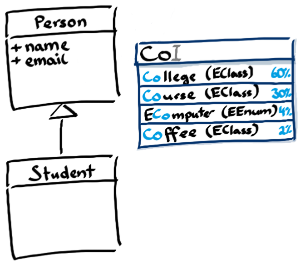


Figure 5: Re-active Recommender

Second, a pro-active recommender system should present information related to the current context

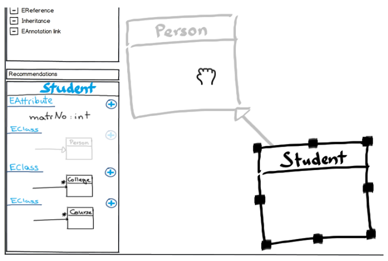


Figure 6: Pro-active Recommender

only. Moreover, the recommendations should be presented in an unobtrusive manner since presenting pro-active information can be easily disturbing. In figure 6, we designed a list of recommendations in a separate window (lower left corner). This list is updated along the context, e.g., if a class is selected or added. A minimalist pro-active GUI can be a text field depicting the status by updating the number of recommendations found. Here, a user needs to request the listing of recommendations explicitly.

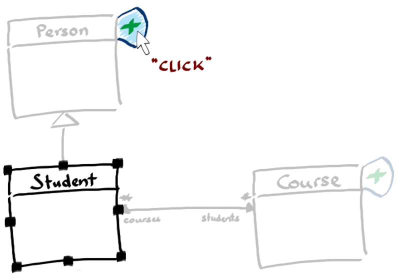


Figure 7: Classes Overlay

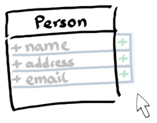


Figure 8: Attributes Overlay

Third, overlays might integrate the user interaction into the canvas (see figure 7 and figure 8). Those are activated automatically or manually by the user. Considering recommendations that alter elements, we need to adapt the overlays to put emphasis on the changes. Mind that this should be used only for re-active recommendations since it most disturbing.

3. How are the recommendations presented? The most common ways to display recommendations are lists and overlays. Lists could hold entries, which are either textual, as shown in figure 5, graphical, or both as depicted in figure 6. Here, one has to consider the amount of recommendations shown in the list. Bollen et al. studied the *choice overload effect* in recommender systems (Bollen et al., 2010). Their findings have implications for the design of recommender system user interfaces; in particular, a recommendation set that contains between five and twenty items works best.

The overlay approach tentatively integrates recommendations into the diagram. For example, figure 7 shows how this might look like and depicts a preview of a possible result. But, there are two problems with this approach. First, the display space is limited, which might hide overlays or might cause overlapping. Second, recommendations that alter or remove elements in the diagram are difficult to stress.

4. How can a recommendation be previewed? We sketched two kinds of recommendation previews. Figure 9 shows a graphical preview as a *thumbnail*, which can either be a box appearing next to the selection or an *outline* window. Further, previews can be overlays as depicted in figure 10. Again, we have to consider display limitations and altered elements.

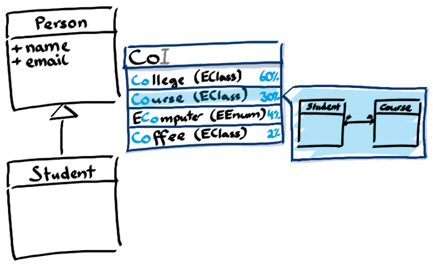


Figure 9: Thumbnail Preview

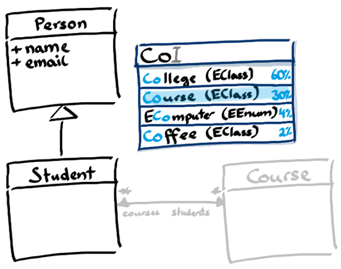


Figure 10: Overlay Preview

5. How does the user pick a recommendation? This question is strongly influenced by the presentation of recommendations. First, lists allow double-click actions (see figure 6) or press-enter-key actions (see figure 5). Second, overlay GUIs allow double-click actions as well, but this interaction is not intuitive for most users. Therefore, a *plus*-button as shown in figure 7 and figure 8 might be the better alternative. In addition, a *minus*-button could indicate remove actions. Additionally, lists might have *plus*-buttons like in figure 6. Last, a list of recommendations in a separate window as shown in figure 6 allows drag-and-drop actions. The advantage over an insertion, which includes an auto-layout, is that the user can place elements properly.

4.2 Guided Interviews

All interviewees had no or little knowledge of recommender systems. After a short introduction, they had a vague idea of recommender systems, but could not transfer this idea to modeling. One of the interviewees said: *"I can hardly imagine recommendations [in modeling], yet."*. But after providing a domain and an example how to recommend classes and relationships, they got a good idea about it. Still, few interviewees were unsure if such a system would help expert modelers. Hence, one of the experts mentioned:

> *"Before I start modeling, I already have a coarse model in mind and know how to proceed. A recommender system would have to know the semantic meaning of what I am modeling to give proper recommendations."*

One of the interviewees noted *"I cannot imagine how recommendation of class or entity names can work."* The suggestion was recommending *"reusable bigger components"*. Additionally, some interviewees were concerned about over-automating modeling:

> *"One should avoid the system to be annoying. [..] Consider the auto correction in today's mobile phones. If the system thinks I misspelled a word and auto correct the word, I have to delete this word and retype. This creates frustration."*

Finally, we exhibited the mock-ups and the interviewees gave hints for improvements. Regarding the preview of recommendations one of the experts argued:

> *"Just a graphical representation [the preview] of recommendations is not sufficient, but is a nice aid. However, a text-only representation is sufficient for me."*

Another interviewee added: *"Overlays [ed., see figure 10] are cool, but can be disturbing. Especially if they are shown in a full screen model."* To sum up the interviews, there were concerns about the use of recommender systems in modeling, but the interviewees were curious about the system. Yet, after we implemented a functional prototype, first of all the expert modelers said they *"badly want this tool: NOW!"*.

4.3 Online Questionnaire

After brainstorming, interviewing, and pretesting the survey, we presented the survey to a broader audience asking to participate. We found great curiosity in such a system in the EMF modeling community. Surprisingly, we got slightly different results compared to the interviews. On the one hand, nearly half of the participants wanted a recommender system for modeling editors right away. On the other hand, a large percentage of the surveyed did not know if they could be able to make use of such a system.

The first result is actually not part of the questionnaire, but it shows how the community responded: We created a thread in the Eclipse-Forum at the modeling community with the title *"An "intelligent" autocomplete function for Ecore diagram editors"* (EMF-Board, 2012). The title did not contain any words like "survey" or "questionnaire" on purpose, since we wanted to test curiosity on this topic. After introducing our research topic, we asked the people to participate in our survey. This post got about 600 views in less than 24 hours, while most other post never get this many views at all. At the end of the survey, this post had about 2400 views. Compared to other posts on the board only "really hot topic"-post get this many views. Thus, we can conclude that the modeling community has a great interest in an autocomplete function or a recommender system for models.

In total, forty (40) beginner, advanced and expert modelers filled out our questionnaire. Each question was answered by thirty-nine (39) participants on average, since, we did not make questions mandatory. As a consequence, we only had twenty-nine (29) entirely completed questionnaires.

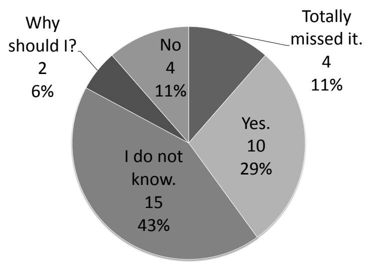


Figure 11: Do you miss a recommender?

Of thirty-nine (39) participants, about 31% consider themselves having a lot of modeling experience, 41% say they have advanced modeling skills and 28% see themselves as beginners. Only three of those polled do not use software tools to design class diagrams. In a comment field they reasoned why they are not using software for modeling.

> *"[Modeling software is] a little bit complex."*

> *"Hand-drawn sketches are usually enough [..]. We draw it digitally only when there are frequent updates, or when we want a permanent documentation."*

Further, only 33% of these participants reuse models. Interestingly, this is a high percentage considering today's weak support to store and retrieve models in libraries. However, only one of the participants uses a team library, the others reuse local files, and no participant uses libraries like MOOGLE (Lucrédio et al., 2008), or ReMoDD (ReMMoD, 2012).

Regarding the section about recommender systems, we first asked if participants miss a recommender system in modeling tools. The majority (about 43%) unsurprisingly stated that they do not know (see figure 11). This seems all natural, since there is no such system in today's modeling software. However, approximately 40% are missing this feature and only 17% do not.

Next, we asked users what they consider as useful recommendations. As one can see in figure 12, all suggested recommendations were highly appreciated, and a lot of participants voted for design patterns as recommendations. Moreover, one participant suggested in a free text field to recommend *"the normal usage of the class"* (ed., how a class is usually defined in a specific context).

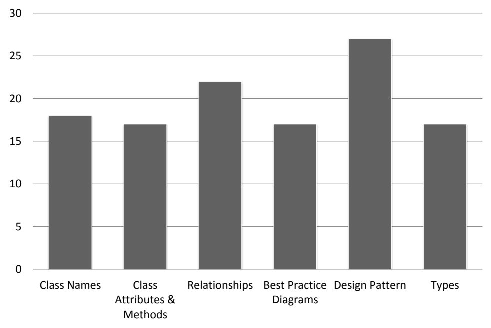


Figure 12: What should be recommended?

In the free-thoughts text field on recommender systems for class diagram editors, we received many suggestions. In the following, we briefly present some of the suggestions. To begin with, one of the participants addressed the interaction with a modeling software and argued: *"Although diagramming [ed., modeling] is a graphical task, a tool should be totally keyboard-friendly for entering speed."*

Moreover, users desired a preview for a recommendation in order to get an idea how the model would look like when the recommendation is applied. To this end, one surveyed user said: *"If the best practices will be recommended, it were nice to have the recommendation as an example in a separate part of the window."*

One participant came up with what we called overlays for class attributes and methods (cf. figure 8).

"The attributes/methods of a class can appear in other color (or better as transparent text), and if I click on it, it will be asked if I wish to add the transparent elements to my class or not."

Summarizing the results of this questionnaire section, we found that a good share did not know if a recommender system would be useful. However, many participants would appreciate such a system. Above all, the participants gave us some great ideas and confirmed other ideas of ours, although we have not yet presented the mock-ups.

Finally, we presented our mock-ups. Before asking them any specific questions related to the mock-ups, we wanted to know how appealing they felt. As we can see in figure 13, most of the participants liked the presented ideas. Moreover, we can see that the Searchbox approach in figure 5, the pro-active approach using the overlays in figure 7 and figure 8, and the overlay previews in figure 10 are the favorite solutions. The pro-active approach within a separate window like in figure 6 was not appealing. However, its drag-and-drop interaction was liked. Last, the screenshot preview in figure 9 was not the first choice.

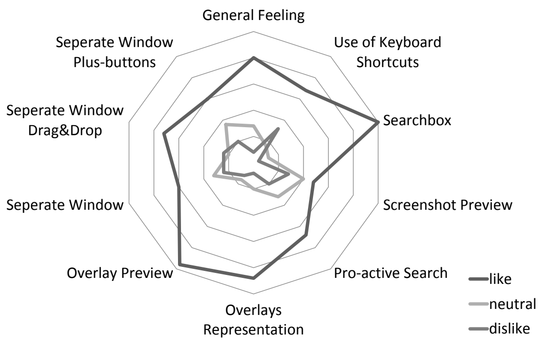


Figure 13: Mock-up Assessments

The most striking result is that a lot of these ideas were liked, but they need to be realized quite differently. Hence, there is not only one solution for a tool realizing all of the ideas at hand, but rather a framework with a flexible architecture, which allows various options for realizations.

Next, we examined what kind of recommendation preview appeared more suitable in small (less or equal 15 elements) and big (more than 16 elements) diagrams. As shown in figure 14 the overlay preview is preferred in small diagrams, but the thumbnail preview is not rejected. However, for big diagrams only a few participants think that the overlay preview is suitable and prefer the thumbnail preview either in an outline window or a box like in figure 9.

Furthermore, we were interested in keyboard friendliness of such a system. To this end, we asked

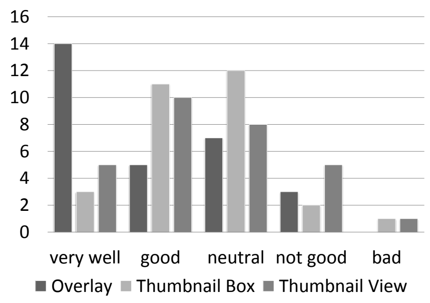


Figure 14: Suitable Previews for Small Diagrams

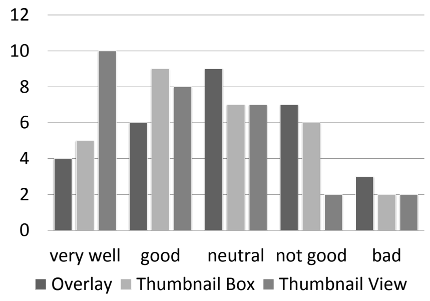


Figure 15: Suitable Previews for Large Diagrams

participants what keyboard shortcuts they expect for what interaction. We found, that 18 out of 29 participants (about 62%) would like to simply start typing and the searchbox should appear. Additionally, the common shortcut CTRL/CMD+Enter was suggested to trigger the recommender system.

Finally, we provided some space for further notes on recommender systems and mock-ups. As confirmed by figure 15, some participants discussed: *"At preview with overlay, you may hide the rest of the class."* And that *"Overlaying UI can be difficult, when there are a lot of classes around, so a separate view [ed., for example as a sub-window] or the screenshot box should just work fine."*

Furthermore, it seems to be a good idea to show the overlay representation (cf. figure 7) only if the user requests recommendations explicitly. One participant stated this as *"At pro-active search with overlay preview, show the recommendation only after typing a keyboard shortcut or a activate button on the class, but not simply at selecting a class."*

There is another thought for a re-active GUI that integrates the searchbox into the window, such that it is always visible: *"I'd expect a Searchbox that is always visible."* To this end, the pro-active GUI in figure 6 could be extended to provide this query field.

Altogether, the participants showed great interest in the system, which one participant put as: *"The UI elements are looking pretty nice for me. Interaction and previews are also OK."* Above all, the findings suggest that there is a great curiosity in the community for such a system.

4.4 Further Ideas

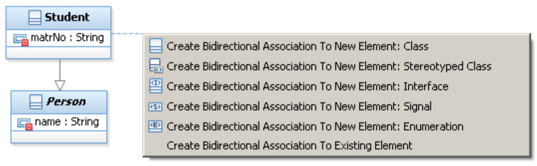
At the end of the interviews and surveys, we offered some space to state further ideas. We summarize:

Almost all of the interviewees proposed a system which might analyze requirement specifications and recommends models based on that. For example, *"If one has a domain model in a context of requirement specifications, [..] one could suggest similar domain models which implements similar requirements."* That makes additional knowledge bases a requirement which do not need to be model libraries.

Moreover, one participant hinted at profile-based recommender systems, i.e., collaborative filtering, and proposed that *"A user should have the possibility to say 'I do not want this recommendation, anymore.'"*. Similarly, a collaborative recommender is to say that *"[The system] should be flexible enough to let users decide about whether something is 'recommendable'."* One can even extend this idea to a social rating in order to opt out bad recommendations. Such a feature is, e.g., implemented in SnipMatch.

Regarding listed recommendations, a participant proposed a filter to classify, because *"It were also good, if it were possible to classify the given recommendations. [..] [They] will be given only for the chosen category [..]."* Quite similarly, an interviewee wanted to categorize the recommendations because there could *"be a lot of recommendations. Thus, you can use accordions or tree views to categorize these."*

Furthermore, a participant yield an idea to provide descriptions for the recommendations because *"you get info about its usage, advantages and so on."* As marginal note, such information should be provided by the underlying model library. However, an architecture of a recommender system should be as flexible to integrate this into the UI.



Student
matrNo : String
Person
name : String
Create Bidirectional Association To New Element: Class
Create Bidirectional Association To New Element: Stereotyped Class
Create Bidirectional Association To New Element: Interface
Create Bidirectional Association To New Element: Signal
Create Bidirectional Association To New Element: Enumeration
Create Bidirectional Association To Existing Element

Figure 16: Association Context Recommendation

Last to mention is that many class diagram editors support drawing an association into empty space. This opens a context menu offering to create new elements at the other end of the association (cf. figure 16). An idea, which has to be examined, is to put recommendations into this menu, e.g., top five classes that are associated with the origin class.

5 SUMMARY AND OUTLOOK

The results of this study show that model recommenders are new grounds. This is best illustrated by the about eighty percent of the participants almost equally split into two groups; who want a model recommender and who do not know. Other than that, the results are significant in three major respects.

First, there are several alternative UIs that suggest different triggering, user interaction, and displaying recommendations. Most of which are reasonable and applicable in different circumstances. Most importantly, the size of recommended models might serve as a deal breaker if they should be displayed in an overlay manner. Moreover, changing the type of the model could change reasoning on alternative UIs.

Second, there seem to be different needs of what items are recommended. In class diagrams, we have seen desires for recommendations of classes, attributes, or complex models. E.g., design patterns were asked for and we are currently investigating on that. Similarly, a model recommender should not be limited to a one back-end. Hence, different recommendation algorithms, each of which implementing another algorithm, need common grounds.

Finally, not only model libraries (e.g., (Ganser and Lichter, 2013)) are requested as data sources for recommendations. In project environments, lots of artifacts are created and could serve as inputs for producing recommendations; e.g., a requirements document. We can use it to help producing recommendations since this can play the role of a user profile in terms of collaborative recommender system. Yet, we are aware of self contradicting requirements specifications. Moreover, the recommender system might be used in diverse modeling tools. There is, therefore, need for multiple recommendation contexts.

All in all, this leads to the need of a flexible and generic architecture. This can bolster research by offering three options for extensions, namely various UIs, algorithms, and contexts. Our realization of this can be found here (Dyck et al., 2014).

REFERENCES

Al Tair, H., Zemerly, M. J., and Al-Qutayri, M. (2012). Architecture for Context-Aware Pro-Active Recommender System. *International Journal Multimedia and Image Processing*, 2(1/2):125–133.

Bennett, J. and Lanning, S. (2007). The Netflix Prize. In *Proceedings of KDD cup and workshop*.

Bobkowska, A. and Reszke, K. (2005). Usability of UML Modeling Tools. In *Conference on Software Engineering: Evolution and Emerging Technologies*.

Bollen, D., Knijnenburg, B. P., Willemsen, M. C., and Graus, M. (2010). Understanding choice overload in recommender systems. In *ACM conference on Recommender Systems*, RecSys '10, USA. ACM.

Bruch, M. (2008). Towards Control-flow Aware Code Recommender Systems. In *3rd International Doctoral Symposium on Empirical Software Engineering 2008*.

Bruch, M. (2012). *IDE 2.0: Leveraging the Wisdom of the Software Engineering Crowds*. PhD thesis.

Dyck, A., Ganser, A., and Lichter, H. (2014). A framework for model recommenders – requirements, architecture and tool support. In *Modelsward 2014, PT*.

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

EMFBoard (2012). EMF Forum: An "intelligent" autocomplete function for Ecore diagram editors.

Felfernig, A., Gordea, S., Jannach, D., Teppan, E., and Zanker, M. (2007). A Short Survey of Recommendation Technologies in Travel and Tourism. *OEGAI Journal*, 25(7):17–22.

Ganser, A. and Lichter, H. (2013). Engineering model recommender foundations - from class completion to model recommendations. In *Modelsward 2013, ESP*.

Happel, H.-J. and Maalej, W. (2008). Potentials and Challenges of Recommendation Systems for Software Development. In *International Workshop on Recommendation Systems for Software Engineering*.

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

Linden, G., Smith, B., and York, J. (2003). Amazon.com recommendations: item-to-item collaborative filtering. *Internet Computing, IEEE*, 7(1):76–80.

Lucrédio, D., De M Fortes, R., and Whittle, J. (2008). MOOGLE: A Model Search Engine. In *Model Driven Engineering Languages and Systems*, LNCS.

Puerta Melguizo, M. C., Boves, L., Deshpande, A., and Ramos, O. M. n. (2007). A Proactive Recommendation System for Writing: Helping Without Disrupting. In *European Conference on Cognitive Ergonomics*.

ReMMoD (2012). http://www.cs.colostate.edu/remodd/v1/.

Resnick, P. and Varian, H. R. (1997). Recommender Systems. *Communications of the ACM*, 40(3):56–58.

Robillard, M., Walker, R., and Zimmermann, T. (2010). Recommendation Systems for Software Engineering. In *IEEE Software*, volume 27, pages 80–86.

Schafer, J. B., Konstan, J., and Riedi, J. (1999). Recommender Systems in E-Commerce. In *ACM Conference on Electronic Commerce*.

SnipMatch (2012). http://www.snipmatch.com/.

Systems, A. R. (2009). http://recsys.acm.org/2009/.

Weimer, M., Karatzoglou, A., and Bruch, M. (2009). Maximum Margin Matrix Factorization for Code Recommendation. In *ACM conference on Recommender systems*, RecSys '09, New York, NY, USA. ACM.