# Engineering Model Recommender Foundations
## *From Class Completion to Model Recommendations*

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

Keywords: Model Completion, Model Recommendation, Model Repository, Model Reuse, MDE, MDD, EMF

Abstract: Reuse has been widely carried out successfully, but not with models in Model Driven Engineering. Reasons seem manifold and conceptual issues and poor tool support are among them. A closer look at the tools available shows that models are often held in repositories which merely exceed versioning and indexing support. But model reuse requires mature approaches and tool support to become successful.
We created a solid conceptual foundation and found recommendations as one solution, which in turn need appropriate data. We engineer these data subsequently and explain our design rationales. In a nutshell, we create a knowledge library comprising of elements which are connected on generic, semantic, and syntactic level. This knowledge library forms an enhanced knowledge graph enabling chain recommendations.

## 1 SETTING THE SCENE

Software development has been becoming more and more complex and increasing "IT-demand" is unlikely to put this to an end. Unsurprisingly, numerous approaches emerged to handle increasing complexity in order to try to ease the challenges in development. One of these approaches is Model-Driven Development (MDD) and it suggests the Unified Modeling Language (UML) as a language to formulate artifacts. The reasons why UML is used might be, because "especially for larger and distributed projects, UML modeling is believed to contribute to shared understanding of the system and more effective communication" (Chaudron et al., 2012).
A closer look at MDD unveils, that it describes a means for designing and developing software at a relatively high level of abstraction. This means, that objects from the real world are mapped to abstract artifacts called models (Muller et al., 2009), (Rodriguez-Priego et al., 2010), (Muller et al., 2012). As a consequence, information needs to be omitted while modeling. This is why modelers often needs to reason if an entity can or does contribute to the overall solution. In other words, to find the appropriate level of abstraction became the major challenge in modeling. But the challenge starts in domain modeling already. If the domain model is not met precisely, the entire system under development is at the verge of failing. Briand et al. investigated this and recommend to "build a domain model as early as possible" (Briand et al., 2012). Another common way to approach development risks and improve quality is reuse. Looking back into computer science history, reuse has been applied for a long time already. It was done by approaching recurring problems by applying conserved solutions or best practices. Functions are probably one of the simplest examples for reuse. Of course, MDD addresses reuse on a conceptual level. Surprisingly, though, only few efforts are undertaken in industries or research to bolster model reuse. Existing approaches barely offer more than simple repository functionality which merely suits a modelers needs.

We believe, that the domain of model reuse is not well understood, and, consequently, approaches and tools offer potential for improvement. Model repositories and model management are available for decades already (sec. 2), but all these do not seem to address the modeler's needs (any more), or we, to the best of our knowledge, are not aware it. It seems, that a well designed knowledge library connecting its elements could bolster model reuse. This is why we would like to fill this gap, by providing the foundations in terms of engineering the domain (sec. 3), modeling the domain (sec. 4), organizing the data (sec. 4.2), enhancing the data (sec. 4.3), and discussing the benefits (sec. 5). These results led to a solid framework in form of an enhanced knowledge graph for tool support which is discussed as an evaluation of the ideas and concepts presented below.

# 2 RELATED STATE OF THE ART

We distinguish model reuse in terms of model repositories and model management, both including retrieval mechanism. This adheres to definitions of repositories in general (IEEE Computer Society and ISO/IEC, 2012). But we intend to go one step further and take into account recommender systems.

## 2.1 Model Repositories

Most approaches in model reuse store models and offer powerful querying. The most prominent systems are contrasted to our system subsequently.
First, MOOGLE, a model search engine, specializes in XMI persisted model files omitting tags and indexing only relevant content (Lucrdio et al., 2010). It distinguishes itself from other model repositories with advanced search options, browsing functionality, and claims to be the most user friendly tool using a website interface. Our system differs from MOOGLE, because we seamlessly integrate our back-end in Eclipse, interlink the stored models and can integrate MOOGLE elements through `URI`s.
Second, ReMoDD is the "Repository for Model Driven Development" from Colorado State University (France et al., 2007). It has mostly documentary purpose and aims to offer models to a community. Hence, the models are not connected as in our system. But, since the data is externally accessible, we can refer to these models via `URI`s.
Third, the "Model Repository", which was developed by University of Leipzig, is slightly different. It uses almost the same technologies as we do but follows a different approach (Uni-Leipzig, 2012). The Model Repository handles model data as graphs and mirrors this entirely in a graph structure. Hence, classes become nodes and associations become edges. In contrast, we handle entire models in a node and use the graph structure as a meta structure which connects the nodes, i.e. the models with each other.
Last, AMOR (Altmanninger et al., 2008), the "adaptable model versioning repository", follows a research approach investigating conflicts between model versions and approaches how to resolve them. It is a closely related project to modelCVS (G. Kappel, 2005), and SMoVer, a semantic model version control system (Altmanninger, 2008). A survey on model versioning approaches summarizes such systems (Altmanninger et al., 2009). Our system is different, since we use the results for our evolution and quality extension and build on top of it, but conflict resolution and versioning are not our primary concerns.

## 2.2 Model Management

Other approaches in model reuse are summarized as model management. This has been under research for a long time already. Hence, a survey from the early nineties (Bharadwaj et al., 1992), which is on mathematical models, already categorizes model management in database-, knowledge-, and graph-based systems. Our system belongs in all of these categories. A glance at generic database model management by Melnik (Melnik, 2004) shows that relationships between models were not considered.
One issue often addressed in model management is checking models while editing, e.g. SmartEMF (Hessellund, 2007). It considers consistency checking, validating operations while editing, and evolution support. We, in contrast, see our approach as editing support in domain modeling and do not consider these aspects except for evolution.
Another issue is categorizing models, as e.g. explained in a facet library (Schmidt et al., 2010). Schmidt et al. introduce a system which supports faceted classification and faceted browsing. This idea is similar to categories as known from web shops, e.g. Amazon. We have such a mechanism as well, but go beyond and interlink our models and offer grouping which is more generic.
Architecturally speaking, the universal repository architecture attempts to handle systems on a more generic level. These attempts have been undertaken in the nineties (Iyengar, 1998) but with different technologies. Petro et al. have build a reuse repository which remains in one domain fostering domain specific software architectures (DSSA) (Petro et al., 1995). We look at a broader, more common perspective, and see this as a special case of our system. What we do not address, though, is mappings between requirements and artifacts yet.
In general, our system was inspired by code completion as offered by JDT in Eclipse. This was recently enhanced by Code Recommenders (Weimer et al., 2009), (Eclipse, 2012a). This is a more clever code completion which includes rankings for code suggestions derived from the editing context. Due to that, recommender systems are related as well as context management (Jannach et al., 2010), (Ricci et al., 2010), (Bettini et al., 2010).

# 3 ENGINEERING FOUNDATIONS

We started our project with a vision which was ignited by looking at content assist in Eclipse. Getting this for classes was the very first idea long ago ...

## 3.1 Model Completion Vision

We transformed our idea of “Class Completion” into a paper prototype (figure 1). It shows a class diagram in the top left corner, which comprises of few classes. Now, the question is how to continue from here? Why not ask a *supporting system* (see center)?

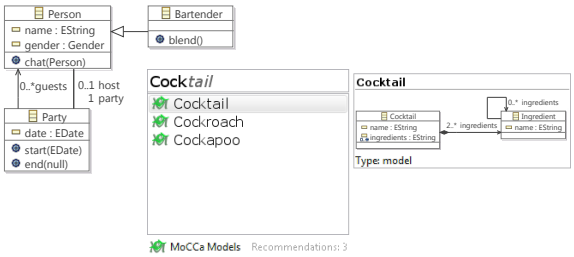


Figure 1: Realization of “Class Completion” Vision

Invoking `Crtl+Space` should open a query bar provided by the *supporting system* and while typing `Cock` the *supporting system* should query a database and provide suggestions in a drop down list. These suggestions should be preview-able by stepping through the list. Placing the selected suggestion into the current canvas should be possible as well. For example, figure 1 shows how the *supporting system* finds three suggestions which are not alphabetically ordered on purpose. This is why figure 1 lists the most related entry *Cocktail* first.
Hence, filling the list needs to consider the environment of the query. In our example, the *Bartender* is the determining element which causes the list showing *Cocktail* atop. This is due to the *supporting system* knowing, that a *Bartender* blends *Cocktail*s and this information is preserved in the database. Consequently, placing the *Cocktail* elements provided by the suggestion should reestablish as much knowledge as possible. Hence, we use the term recommendation from here on.

## 3.2 Model Completion Use Cases

Starting with a vision to get “class completion”, we extended the idea to “model completion” and identified the use cases depicted in figure 2. It only shows an excerpt of the actual diagram but everything necessary to understand the requirements.
In more detail, the `Store Model` use case allows a `Modeler` to extract some elements from a class diagram and store them in the system. While doing so, the `Modeler` can describe the model by setting a name, purpose, description, and so on. Moreover, the `Modeler` can link the model with other models.

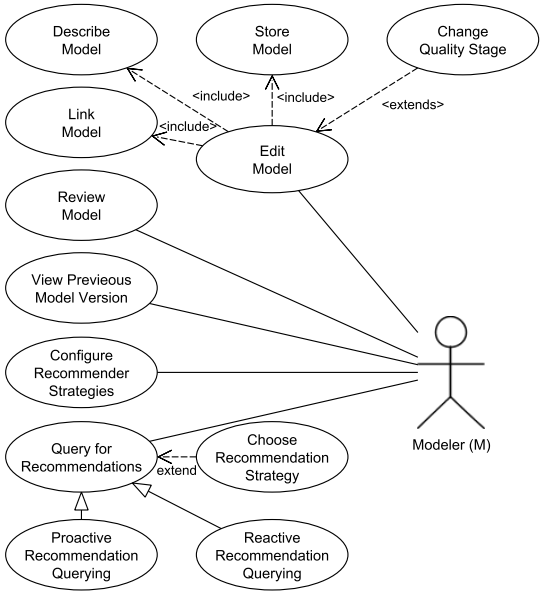


Figure 2: Excerpt of Use Case Diagram

The `Query Recommendations` use case describes how the system reacts on queries. But, this requires configured recommendation strategies, which can be achieved through the `Configure Recommender Strategies` use case. Since, the architecture of the recommender system is not the main focus of this paper, we do not go into further details here. Essentially, the configuration and querying allows multiple strategies for multiple data sources providing recommendations for multiple editors on a flexible architecture. This means, the architecture is open for other services. For example, we implemented a recommender strategy which bases on WordWebOnline (Wordweb Software, 2012). Querying displays the parsed website in the drop down menu and, e.g. confirming a ”House” as ”type of Building” inserts an abstract class *Building* and a class *House* on the canvas which are connected by inheritance.
Note, that the use case `Edit Model` is most essential for this paper, but the other use cases contributed to its requirements; often in a sense that they require a specialization of `Edit Model`.
Moreover, we analyzed users and actors which are not all shown in figure 2. First, `Domain Experts` model the domain concepts, describe, and review models. The `Programmer`s view and review models since they are the ones who tailor the generated source code in the end. Last, `Modeler`s are involved in almost every use case.

## 3.3 Model Completion Architecture

We subdivided the model completion system into five components as illustrated in figure 3. It comprises of a core, which is the knowledge library, and of surrounding components; each of which maps to an Eclipse Feature. Before we go into details regarding the knowledge library, we briefly explain the other components and how they evolved. This will help understanding design rationales.

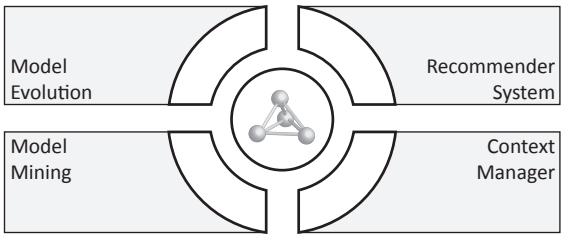


Figure 3: Architectural Overview

We build the components as follows. First, we added storing and querying support. Second, we extended the querying to a recommender system. Roughly at the same time, we added evolution aspects, so we could add model mining, and the context manager.

The simple querying was quickly extended to a flexible recommender system because our knowledge library offered much more than just exporting the key word matches. In fact, the underlying knowledge graph enabled us right away to attach properties on edges and nodes. But to get the most out of such an enhanced knowledge graph requires a recommender system which is configurable in at least two ways. First, multiple recommender strategies are needed to serve different recommendation requirements, and, second, multiple user interface support is needed, so recommendations can be applied in different editors.
The third way to configure the recommender system component is provided by the context management. It manages the query and insert contexts which can be taken into account by the recommender strategies. A query contexts could be an editor, a position, a graph neighbor, or a domain glossary. The editor is relevant, because applying a graphical recommendation into a textual editor must be prohibited. Moreover, a class must not be inserted into the name field of a class. Instead, it is desirable that recommendations are adjusted according to neighbor models from previously inserted models or according to a project glossary.

The simple storing was enhanced to model mining. The initial storing offered slicing for class diagrams preserving relationships in the knowledge library. For example, in our vision (cf. figure 1) imagine an inserted `Cocktail` and a restored relationship between `Bartender` and `Cocktail`. The storing mechanism could slice this into two models, storing them in the knowledge library, while preserving the information about the connection between `Bartender` and `Cocktail`. This is fine, but the knowledge library was not involved except for storing. Enhancing the storing to model mining takes this into account.
Since reusability is barely met at once, our evolution component enhances the knowledge library with evolution and quality aspects. Every model stored is put under evolution and quality control, and can evolve from a vague stage until it becomes deprecated. Without going into further detail, we have three evolution stages which are bound to quality gates. Those are assured by quick reviews and metrics.

Our incremental way building the system derived from the architecture shown in figure 3 proved very reasonable. This way our surrounding components validated and evaluated our design decision let alone the user studies we carried out.

## 3.4 Model Completion Concepts

From the conceptual point of view, we started "Model Completion" as a reuse approach. Beginning with an existing class diagram it should be possible to slice models or extract reusable parts in terms of sets of classes and relationships. In terms of reuse this is a copy-and-modify approach because we do not expect modelers to use the recommendations as they are. For example, a `Cocktail` as shown above could have pricing information if used as part of a menu.
Moreover, we intend extracted models to preserve external connectors as they are stored in an enhanced knowledge graph, if possible. In the example mentioned above, we extracted two models (`Party`, `Cocktail`) from a model containing five classes. These two models would represent two nodes in a graph which are connected with an edge. The nodes would hold the information about the models and the edge would hold the information about the relationship. This means, we use a graph and add properties to nodes and edges. Conceptually speaking this forms an unidirectional property graph, and becomes a weighted graph at the time of the recommendations.

From a technological point of view, we started "Model Completion" since ecore files are commonly used in EMF for code generation but they are often modified in the ecore diagram editor shipped with the Ecore Tools (cf. figure 1). This is why EMF (Steinberg et al., 2009) and the Ecore Tools (Eclipse, 2012b) are our basic dependencies which were given.

# 4 REASONING KNOWLEDGE LIBRARY ELEMENTS

A usual knowledge library comprises of basic building blocks and meta information. Our basic building blocks are `Library Elements`. They can contain `MetaInformation` which is designed as the central extension point for the `Library Elements`. We did so, because this information is supposedly generated separately in components and then linked to `MetaInformation` through extension points.

## 4.1 Isolated Library Elements

Storing a single element in our knowledge library results in a single `LibraryElement` comprising of properties (see figure 4). A `name` and a list of files, which holds at least one file, are compulsory. Providing an `owner` is optional. The `files` are defined as a list of `URIs`, since this makes resource handling in EMF much easier.

The reason why we needed to support different `files` per `LibraryElement` was initially caused by the ecore tools, since they keep the content (i.e. `.ecore`-files) separate from the layout information (i.e. `.ecorediag`-files). Moreover, we did not want to burden our server to generate previews for class diagrams on the fly. This is why we choose to store `png`-files which are loaded for previews as shown in figure 1. A nice side effect of this design decision is, that the preview `png`-files might be manipulated separately for e.g. highlighting certain aspects in a preview. This, to our mind, outweighs the manual maintenance of previews. Anyhow, previews are a specialty to `Models` as the method `getPreview()` implies. Other kinds of `LibraryElements` are not implemented so far, but textual `LibraryElements` would be a very reasonable extension. They could keep textual description of class diagrams.

For every extension which `LibraryElements` might face, we want to rely on the provided libraries and this is why we do not store related files in the database but in the file system. This even allows distributing file locations among servers using `URIs`. Moreover, our indexing becomes much more flexible since we can make use of the file's native format and implement new indexing strategy leveraging the distributors libraries.

Currently, we use the ecore tools to extract data and put them in our indexing. Hence, we can not only query for `name` and `owner`, but class names, attributes, associations and so on.

So far the knowledge library distinguishes itself from every other system available, just because it supports

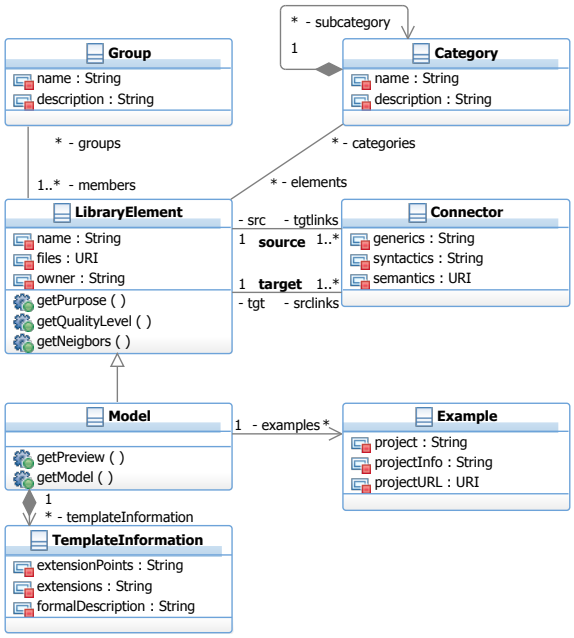


Figure 4: Basic Building Blocks and Graph Support

indexing and querying through strategies. But this is not enough to foster multiple and complex recommender strategies. Consequently, we enhanced the structure developed so far and interlinked, grouped and structured `LibraryElemens`.

## 4.2 Connected Library Elements

There are two different approaches to interlink `LibraryElements`. First, there are meta connectors like grouping and category mechanism which associate `LibraryElements` to meta concepts in order to arrange them indirectly. Second, there are concrete connectors between `LibraryElements` which establish immediate relationships between `LibraryElements`. Figure 4 depicts, how these elements are related. Mind, that it makes perfect sense to generalize these connectors at a conceptual level, but this does not reflect into the modeled domain, i.e. there is no super-class in figure 4.

The meta connectors are `Group` and `Category`. First of which arranges one or more `LibraryElement` to a set of elements. Since several `LibraryElements` often occur in groups, though they are independent, we introduced this concept. For example, we could have a class diagram which models a *Person*. This, as a `Model`, could be part of a group with either a *BankAccount* or *ClinicalRecord* or both. Clearly, a *Person* becomes a client or patient, which will be described below. But altogether, these `Models` do not belong to one `Group` or `Category`.

The other meta connector, which is `Category`, also groups `LibraryElements`, but as intuitively implied by the name. Taking the examples from above, the *ClinicalRecord* would belong to a `Category` *health-care* and the *BankAccount* would belong to a `Category` *financing*. Finding categorically related, but not grouped `LibraryElements` is the purpose of `Categories`.

The concrete connectors have three different natures, namely generic, syntactic, and semantic. Generic connectors establish a link between two `LibraryElements` without providing any more details. This is reasonable, if the details are unknown or at least one of the `LibraryElements` is held outside of the knowledge library. In case, that the details of the connector are unknown, the generic connector can be seen as "these two models have been used together in the past". In our *Party*, *Cocktail* example this would mean, that the information about the association between *Bartender* and *Cocktail* is not in the knowledge library.
The second nature of concrete connectors is called syntactic and provides additional information about a connection between two `LibraryElements`. In our above mentioned example, the syntactic information would be the association, which would be attached to the connector. Hence, placing the *Party*-`Model` first, getting the *Cocktail*-`Model` recommended and placing it, would, this time, reestablish the association between *Bartender* and *Cocktail* and look like figure 5. In fact, syntactical connectors allow much more than that. They can hold entire bridge models. As an example, imagine a *person* as a *customer* in one domain and as a *patient* in another domain as mentioned above. Then it makes not much sense to adjust the *person*-`Model`, but instead place an adapter, facade, or proxy in the syntactical connector. This model then actually holds all the "dangling" references.

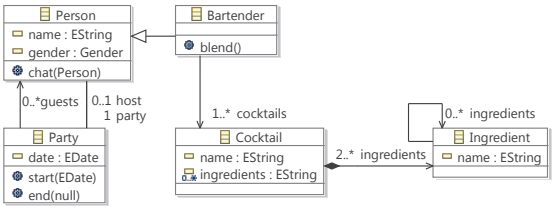


Figure 5: Model Restored with Syntactic Connector Nature

The last nature of concrete connectors is of type semantic and puts design rationales in focus. Therefore, the purpose of this connector is mostly documentary, but not only. For example, taking our above example, it might be patently obvious to the modeler, who has put this in the knowledge library, that the *Clinical-Record*-`Model` is connected to the *Person*-`Model` via a proxy-adapter. That means, the technical connector adapts the *person* and at the same time prohibits access to certain information in the *Person*-`Model`. But is that obvious to a modeler who wants to reuse this information and has never seen this model before? This is why we need to enhance connectors with this information in the mining process or manually.

A closer look at the concrete connectors unveils, that they are easy extendable in graph structures to hyper-edges. This means, generic connectors become multi-generic connectors, syntactical connectors become multi-syntactical connectors and so on. But what does this mean and what are the benefits and drawbacks?
Hyper-edges are certainly needed if we extend our example from our vision in figure 1 with a *billing*-`Model`. This *billing*-`Model` could have a *printReceipt()* method which requires a *Guest* and a list of *Cocktail*s. This means, that the *billing*-`Model` has two dependencies from one element to two different models but a property graph allows edges of cardinality one only. A hyper-edge would solve this problem and simply create a hyper-edge which comprises of a set of nodes which are supposedly connected.

Putting all these information together, the developed knowledge library offers a very flexible and powerful source for recommender strategies. This is due to, first, data, second, grouping and categorizing, third, the connector concept, and, fourth, the traversal mechanism which can benefit from each as needed. But this needs the concepts mapped to a property graph structure which is straight forward. Unfortunately, the graph structure with its nodes and edges are only one side of the coin, because the information content of the nodes is rather simple so far. Due to that, we enhanced the `LibraryElements`.

## 4.3 Enhanced Library Elements

Models stored for reuse as `LibraryElements` in a property graph are a good start, but this structure proofed "only good" for recommender strategies and other components. Therefore, we introduced `MetaInformation` which can be contained in `LibraryElements`. This means, `MetaInformation` serves as an extension point for extra information.
As an example, we extended `MetaInformation` with evolution and quality aspects as shown in figure 6. Note, that some of the information defined in `MetaInformation` is accessible from `LibraryElements`; e.g. through `getQualityLevel()` or `getPurpose()`. Without going into further detail, we use the `purpose` as

a lightweight specification mechanism, so we have an anchor for reviews and quality aspects as mentioned in section 3.3. Last of which even introduces `VersionInfo` and attaches semantics in terms of quality levels to `LibraryElements`.

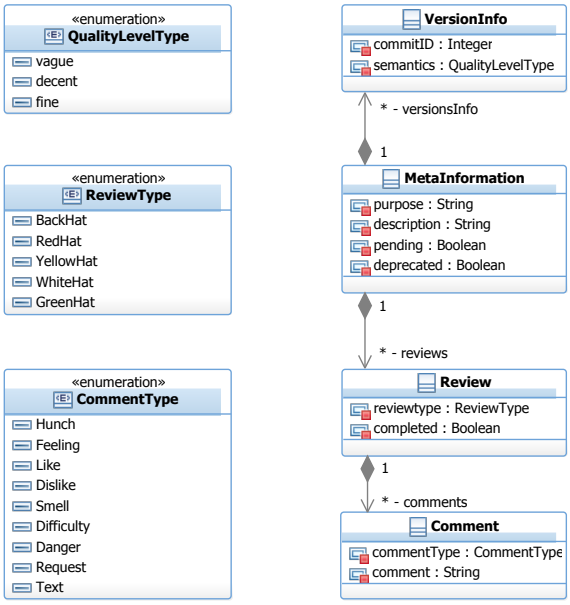


Figure 6: Additional Meta Information

Other extensions we created are `Examples` and `TemplateInformation` as depicted in figure 4. While `Examples` are intended to guide users how to apply a `Model`, `TemplateInformation` is intended to generalize `Models` for recommendations. Simple placeholder, linked variables, extension points, extensions, and guided completion in terms of extending, pruning or adapting are currently under research for the template mechanism.

# 5 DISCUSSING BENEFITS

The developed knowledge library supports "Model Completion" in multiple ways. First, our indexing strategies support querying for `name`, `description`, and `purpose` attributes. Moreover, `MetaInformation`, `description`, `LibraryElement files`-content are indexed. Consequently, depending on how the query is formulated, certain indexed information can be omitted. This is independent of the graph structure, but we can limit query results to certain ranges by that. This limitation of queries becomes very handy during model mining, because a query limited to class names can unveil "known" information. Moreover, this is very useful if a query context limits to recommendations which are actually applicable in the editing context.

Other benefits arise due to the graph structure and allow follow-up or chain recommendations. Going back to our initial example from figure 1, we think of our knowledge library containing the *Cocktail*-`Model` with a linked *Party*-`Model`. Starting from an empty canvas, invoking `Ctrl+Space` and typing `Party`, confirming the `Model` recommendation, gaining the canvas as in figure 1 a chain recommendation could immediately insert the *Cocktail*-`Model`, granted there is a very high likelihood, that this `Model` usually "follows" in a modeling process.

But the graph structure as designed even supports context management for recommender strategies. For example, the example from our vision in figure 1 could be completed with the *Cocktail*-`Model` and keep the neighbors of the previously inserted `LibraryElements` in an editing context. This would hold all neighbors from the *Cocktail*-Model in the first level of the history context and all the neighbors of the *Party*-`Model` in the second level of the history context. Now querying the recommender strategies could consider the query history.

# 6 SUMMARY & FUTURE WORK

Starting with a vision of "Class Completion" as shown in figure 1, we analyzed the domain, shifted the vision to "Model Completion", build use cases and pointed out the major requirements. This lead to a coarse grain architecture (figure 3) comprising of a core, which is the knowledge library and surrounding components. These were taken into account, since they had either impact on the design of the core or they evaluated and validated the conceptual design of the core.

The core itself was explained in detail by, first, pointing out how isolated library elements are modeled, and then extending them to interlinked library elements. There we introduced different natures of connectors, i.e. generic, syntactic, semantic. Finally, we enhanced the interlinked elements with additional concepts which foster recommender strategies and ease their design.

The work in progress comprises of the template mechanism, context management, and model mining. We do so, in order to make the system as seamlessly integrated in a development environment and support quick starts as much as possible. Therefore, query contexts managed by context management for recommender strategies are as much of vital importance for usability as model mining is for usefulness.

## ACKNOWLEDGEMENTS

We would like to thank all our reviewers for their comments! We would also like to thank Felix Bohuschke, Andrej Dyck, Christian Fuchs, Ruslan Ragimov, and Alexander Roth for their contributions.

## REFERENCES

Altmanninger, K. (2008). Models in Conflict Towards a Semantically Enhanced Version Control System for Models. In Giese, H., editor, *Models in Software Engineering*, volume 5002 of *LNCS*, pages 293–304. Springer Berlin / Heidelberg.

Altmanninger, K., Kusel, A., Retschitzegger, W., Seidl, M., and Wimmer, M. (2008). AMOR Towards Adaptable Model Versioning. http://www.modelversioning.org.

Altmanninger, K., Seidl, M., and Wimmer, M. (2009). A survey on model versioning approaches. volume 5, pages 271–304.

Bettini, C., Brdiczka, O., Henricksen, K., Indulska, J., Nicklas, D., Ranganathan, A., and Riboni, D. (2010). A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing*, 6(2):161 – 180.

Bharadwaj, A., Choobineh, J., Lo, A., and Shetty, B. (1992). Model management systems: A survey. *Annals of Operations Research*, 38:17–67. 10.1007/BF02283650.

Briand, L., Falessi, D., Nejati, S., Sabetzadeh, M., and Yue, T. (2012). Research-Based Innovation: A Tale of Three Projects in Model-Driven Engineering. In France, R., Kazmeier, J., Breu, R., and Atkinson, C., editors, *Model Driven Engineering Languages and Systems*, volume 7590 of *LNCS*, pages 793–809. Springer Berlin / Heidelberg.

Chaudron, M., Heijstek, W., and Nugroho, A. (2012). How effective is UML modeling ? *Software and Systems Modeling*, pages 1–10.

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

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

France, R., Bieman, J., and Cheng, B. (2007). Repository for Model Driven Development (ReMoDD). In Kuehne, T., editor, *Models in Software Engineering*, volume 4364 of *LNCS*, pages 311–317. Springer Berlin / Heidelberg.

G. Kappel, G. Kramler, E. K. T. R. W. R. W. S. (2005). *ModelCVS - A Semantic Infrastructure for Model-based Tool Integration*. Technical Report, Johannes Kepler University of Linz and Vienna University of Technology.

Hessellund, A. (2007). SmartEMF: guidance in modeling tools. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 945–946, New York, NY, USA. ACM.

IEEE Computer Society and ISO/IEC (2012). SEVOCAB: Software and Systems Engineering Vocabulary. http://pascal.computer.org/sev_display/index.action.

Iyengar, S. (1998). A universal repository architecture using the OMG UML and MOF. In *Enterprise Distributed Object Computing Workshop, 1998. EDOC '98. Proceedings. Second International*, pages 35 –44.

Jannach, D., Zanker, M., Felfernig, A., and Friedrich, G. (2010). *Recommender Systems: An Introduction*. Cambridge University Press.

Lucrdio, D., de M. Fortes, R., and Whittle, J. (2010). MOOGLE: a metamodel-based model search engine. *Software and Systems Modeling*, 11:183–208.

Melnik, S. (2004). *Generic Model Management*, volume 2967 of *LNCS*. Springer Berlin / Heidelberg.

Muller, P.-A., Fondement, F., and Baudry, B. (2009). Modeling modeling. In Schuerr, Andy and Selic, Bran, editor, *Model Driven Engineering Languages and Systems*, volume 5795 of *LNCS*, pages 2–16. Springer Berlin / Heidelberg.

Muller, P.-A., Fondement, F., Baudry, B., and Combemale, B. (2012). Modeling modeling modeling. *Software and Systems Modeling*, 11:347–359.

Petro, J., Fotta, M., and Weisman, D. (1995). Model-based reuse repositories-concepts and experience. In *Computer-Aided Software Engineering, 1995. Proceedings., Seventh International Workshop on*, pages 60 –69.

Ricci, F., Rokach, L., Shapira, B., and Kantor, P. (2010). *Recommender Systems Handbook*. Springer.

Rodriguez-Priego, E., Garca-Izquierdo, F., and Rubio, n. (2010). Modeling Issues: a Survival Guide for a Non-expert Modeler. In Petriu, D., Rouquette, N., and Haugen, y., editors, *Model Driven Engineering Languages and Systems*, volume 6395 of *LNCS*, pages 361–375. Springer Berlin / Heidelberg.

Schmidt, M., Polowinski, J., Johannes, J., and Fernndez, M. (2010). An Integrated Facet-Based Library for Arbitrary Software Components. In Kuehne, T., Selic, B., Gervais, M.-P., and Terrier, F., editors, *Modelling Foundations and Applications*, volume 6138 of *LNCS*, pages 261–276. Springer Berlin / Heidelberg.

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

Uni-Leipzig (2012). Eclipse Model Repository. http://modelrepository.sourceforge.net/.

Weimer, M., Karatzoglou, A., and Bruch, M. (2009). Maximum margin matrix factorization for code recommendation. In *Proceedings of the third ACM conference on Recommender systems*, RecSys '09, pages 309–312, New York, NY, USA. ACM.

Wordweb Software (2012). WordWeb Online Dictionary and Thesaurus. http://www.wordwebonline.com/.