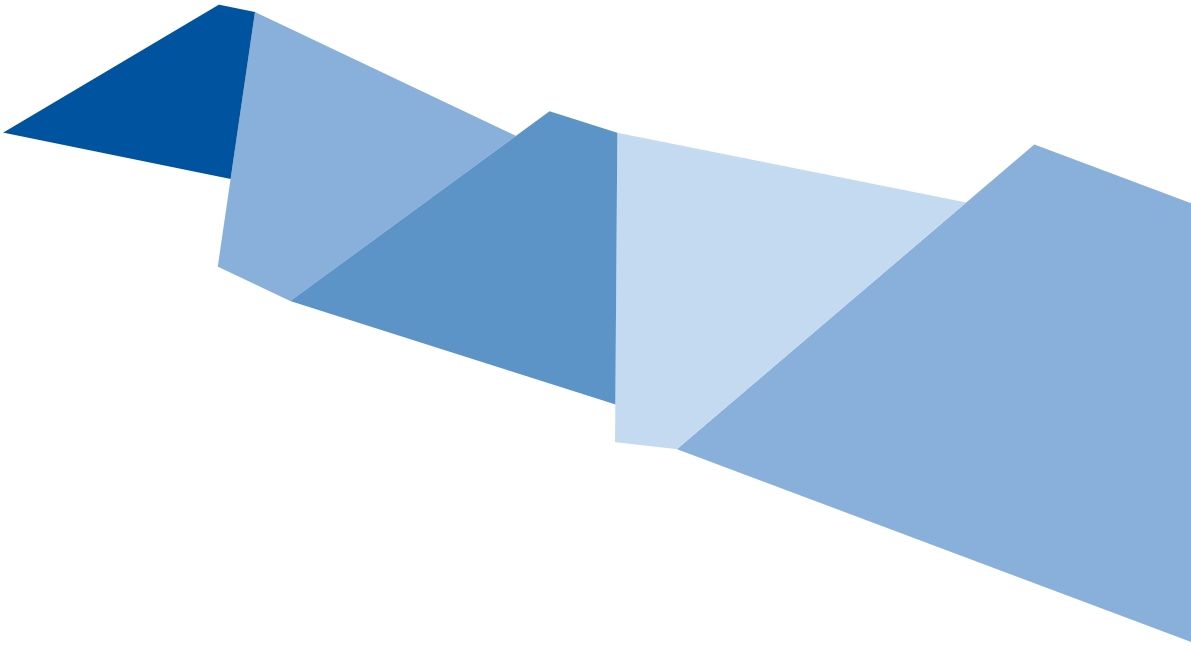


Operation-Based Model Recommenders

Andreas Ganser

Department of Computer Science

Technical Report

Aachener Informatik-Berichte (AIB) | ISSN 0935-3232 | AIB-2018-04
RWTH Aachen University | Department of Computer Science | May 2018

Operation-Based Model Recommenders

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
RWTH Aachen University zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Andreas Ganser
aus Mönchengladbach

Berichter: Universitätsprofessor Dr. rer. nat. Horst Lichter
Universitätsprofessor Dr.-Ing. habil. Matthias Riebisch

Tag der mündlichen Prüfung: 6. November 2017

Die Druckfassung dieser Dissertation ist unter der ISBN 978-3-8440-5946-5 erschienen.

YOU

Abstract

"Reuse is Boring!" states the title of the introductory chapter to IEEE Standard 1517, *Software Reuse Processes*, published at the turn of the century. Later, it states that "[f]or years, we have been hearing about the benefit reuse offers, but have yet to see them realized in practice." This is certainly debatable as a general statement, and success stories in the early 1980s counter this, at least for source code. Still, there must be some reason for such a harsh statement while disregarding types of reuse; after all, loops, methods, and classes are forms of reuse.

For higher-level reuse, i.e., for activities from the preservation until the reutilization of knowledge, we can firstly say that activities involved in reuse do not pay off immediately, but only in the long run. Even worse, all these activities are generally considered tedious, because they expose no immediate benefit. Thus, starting with the harvesting of knowledge and storage for later reuse, continuing with looking for suitable harvested knowledge, i.e., retrieving it, and finishing with reusing (or reutilizing) it, these activities are perceived as rather unappealing. Fortunately, integrated development environments for lower-level reuse, e.g., source code, have already demonstrated how to approach this using completion mechanisms that foster enhanced querying and recommender systems. This places the experience of whole communities at the fingertips of every programmer. Yet, there is no such support for modeling.

Instead, in cases of higher-level reuse, i.e., for activities from harvesting until the reutilization of knowledge, we can secondly state that modelers need to deal with often unorganized information overflow. Fortunately, the abovementioned approaches can help, but need proper information organization. This is known as the challenge of data representation, and can be addressed using a combination of well-suited information representation with a clever retrieval mechanism to enable model reuse that is tightly embedded in tooling.

Further, for higher-level reuse, we can thirdly note that recommended models should be of good quality. Hence, guided assurance in terms of the evolution of harvested models completes the picture of model reuse activities.

Simply put, the abovementioned points are commonly considered to be unappealing activities dealing with challenges denoted by representation, harvesting, evolution, and retrieval. These challenges shall be addressed subsequently. Eventually, we contribute an approach tailored for modeling with UML or models akin to class diagrams, and this approach turns out to be a knowledge-based recommender system based on property graphs and metagraphs suitable for a broader scope. Further, we provide a cookbook for developing such a system, which includes schema for model recommendation production for operation-based model recommenders based on our deployment experiences with HERMES. As we are taking into account contextual information monitored as modeling operations, this too could be denoted as an operation-and-knowledge-based recommender system that (semi-)automates tedious activities.

Kurzdarstellung

Ein einleitendes Kapitel des IEEE Standards 1517 *Softwarewiederverwendungsprozesse* um die Jahrtausendwende heißt „Wiederverwendung ist langweilig!" und später setzt es fort, dass wir „seit Jahren von den Vorteilen hören, die Wiederverwendung bietet und doch muss die Praxis sie erst noch zeigen." Sicherlich ist das als generelle Aussage diskutabel und Berichte aus den frühen Achtzigern zeigen zumindest Erfolge bei Quelltexten. Dennoch muss es einen Grund für solch eine herbe Aussage geben, selbst wenn gewisse Typen von Wiederverwendung ignoriert werden; Immerhin sind Schleifen, Methoden, Klassen und dergleichen auch Formen von Wiederverwendung.
Für Wiederverwendung auf höherem Abstraktionsniveau, d.h. für alle Tätigkeiten Wissen zu konservieren bis hin es als solches wieder einzusetzen, können wir erstens feststellen, dass beteiligte Tätigkeiten sich nicht kurz- sondern nur langfristig lohnen. Es ist gar so, dass sie gewöhnlich als lästig erachtet werden, weil sie keinen direkten Vorteil liefern. Folglich werden diese Tätigkeiten, die mit dem Ernten von Wissen zwecks späterer Verwendung beginnen, sich mit dem Wiederauffinden solches fortsetzen und beim Wiedereinsetzen münden, als eher uninteressant wahrgenommen. Glücklicherweise gibt es integrierte Entwicklungsumgebungen für Wiederverwendung auf niederem Abstraktionsniveau, beispielsweise Quelltext, die veranschaulichen, wie Wiederverwendungsmechanismen mittels spezieller Anfragen und Empfehlungssystemen funktionieren. Damit hat ein Programmierer die Erfahrung von Entwicklergemeinschaften stets unmittelbar zur Hand. Für Modellierer gibt es jedoch nichts Vergleichbares.
Stattdessen können wir zweitens für Wiederverwendung auf höherem Abstraktionsniveau, also alle Tätigkeiten vom Wissen-Ernten bis hin zum Wiedereinsetzen, sagen, dass Modellierer oft mit chaotischem Informationsüberfluss fertig werden müssen. Glücklicherweise können die obigen Ansätze helfen; benötigen aber angemessene Organisation. Diese, typischerweise als Aufgabe der Datendarstellung bekannte Situation, kann nun mit einer Mischung aus wohlstrukturierten Informationen zusammen mit klugen Instrumenten zwecks Wiederfinden angegangen werden und ermöglicht Modellwiederverwendung eng eingebunden in Modellierungswerkzeuge.
Darüber hinaus können wir drittens anmerken, dass empfohlene Modelle von gute Qualität sein sollten. Deshalb vervollständigt angeleitete Betreuung im Sinne von Evolution der geernteten Modelle die Wiederverwendungstätigkeiten.
Nachfolgend gehen wir auf die oben genannten Punkte ein, die üblicherweise als uninteressante Tätigkeiten wahrgenommen werden und die wir einfach gesprochen als Aufgaben hinsichtlich Darstellung, Ernten, Evolution und Wiederauffinden bezeichnen. Letztlich führt das dazu, dass wir einen Ansatz zugeschnitten für UML Modellierung ähnlich zu Klassendiagramme einbringen, der ein wissensbasiertes Empfehlungssystem für Attribut- und Metagrafen umsetzt; aber generischer ist. Zudem stellen wir auf Grundlage unserer Erfahrungen mit HERMES eine Kochanleitung zur Entwicklung solcher bereit, die Schema für Erzeugung von Modellempfehlungen für wissensbasierte Modellempfehlungssysteme enthält. Diese könnten wir letztlich operations- und wissensbasierte Empfehlungssysteme nennen, da sie Umgebungsinformationen in Form von Operationssequenzen betrachten und so lästige Tätigkeiten, sofern sinnvoll, (teil-)automatisieren.

Acknowledgment

The course of this text covers findings from two projects and neglects that they were achieved during a long process and by a tremendous team effort. I would ask whomever has contributed but is not mentioned here to pardon my mistake and accept my sincerest apologies.

First and foremost, my supervisor Universitätsprofessor Dr. rer. nat. Horst Lichter offered me the chance to undertake the endeavor of this research in his department, Research Group Software Construction. I found myself extremely lucky to have an advisor who never got tired of believing in this project whilst providing freedom beyond belief. I also want to thank Universitätsprofessor Dr.-Ing. habil. Matthias Riebisch for being my secondary advisor and providing countless productive suggestions and feedback.

Furthermore, I would like to thank my student workers, bachelor's, master's, and diploma students for their contributions to the HERMES tool. Their efforts and dedication always pushed forward the current state and had an impact on various concepts and realizations. In its final stage, HERMES has the fingerprints of Felix Bohuschke, Stefan Dollase, Andrej Dyck, Christian Fuchs, Niklas Franken, David Mularski, Ramya Nagarajan, Junior Lekane Nimpa, Ruslan Ragimov, Dr. Alexander Roth, Daniel Schiller, Nils Sewing, and Tran Ngoc Viet. Moreover, Roland Hildebrandt, Stefan Hurtz, Christian Kuhl, and Thanabordee Thanarukvudhikorn, though not directly related to these projects, contributed in numerous ways.

I also want to express my sincerest gratitude to my former colleagues Andrej Dyck, Muhammad Firdaus Harun, Dr. Veit Hoffmann, Dr. Simona Jeners, Ana Nicolaescu, Dr. Alexander Nyßen, Malek Obaid, Dr. Chayakorn Piyabunditkul, Dr. Holger Schackmann, Dr. Tanya Sattaya-aphitan, and Dr. Matthias Vianden. They always established a productive, friendly, and encouraging atmosphere at Research Group Software Construction.

In every working group, there are magical fairies doing a job beyond price. We, at Research Group Software Construction, had Bärbel Kronewetter and Marion Zinner backing us up all the time. No matter whether we needed a breakfast during busy times, a helping hand when things got out of hand, or a shoulder to cry on, they were always there for each and every one of us.

Last, but not least, I would like to thank my brother, grandfather, parents, and methyltheobromine for always having my back. You had much more impact on this result than you could possibly imagine and I could possibly express in gratitude.

Andreas Ganser

Contents

1 Approaching Model Reuse

> Nothing in this world that's worth having comes easy...
>
> Robert Kelso, M.D.

Contents

It is quite common to consult relatives, friends, and family on big decisions, and their recommendations are usually held in high regard. Their advice is sought, after all, because it is assumed that they know the advice-seeker well and can advise on what is best for him/her. However, this behavior of "seeking advice from somebody" is not limited to life-changing decisions such as whether to change jobs or get married—it often happens on other occasions and might include other circles of people.

These people can be domain experts, who can help in advice-seeking situations by leveraging their subject-matter knowledge. Sometimes, such as for a dental, legal, or medical expert, only they can tell whether dental treatment is necessary, if there is a chance of winning a lawsuit or making an out-of-court settlement, or if it really makes sense to approach an illness with a particular treatment; after all, they are domain experts and get paid for such consultations. In other advice-seeking situations, experts are paid indirectly, e.g., when purchasing certain products.

Picture a shopping experience in a specialized shop: a retailer offers recommendations on products, and a good retailer will try to sell a suitable package instead of a sole product. This means the product will be accompanied by recommended convenience goods, which together make up the package. Thus, hopefully, the customer is happy with, e.g., a new digital camera package, because it also comprises a memory card and a battery, so the customer can immediately operate the camera. The dealer knew from experience that these complementary products would be necessary and appropriate, i.e., that a particular memory card and particular battery would work with a specific camera.

Moreover, a good retailer will have tried to sell the "most suitable" package in the first place, and will have done so by evaluating the customer's needs. The final package will be a result of the sales pitch, but the chances are that products that have previously been purchased by this and other customers will also have been helpful in this regard. In a nutshell, the retailer knows about the products, their relationships, customers, and purchase histories. In other words, the retailer can make use of knowledge and experience in sales situations, hopefully to the benefit of the customer and the shop.

The subsequent text transfers the ideas of "how to produce recommendations" to software engineering, or more precisely, modeling as an attempt at modeling support.

Today, web shops try to simulate the recommendation aspect of retailers. Hence, the customer evaluation mentioned above is done by computer systems. These systems use modeled relationships between products, but bring about certain obstacles. For example, computer systems have no means (yet) for face-to-face communication. This means a customer cannot be assessed on the nonverbal level of communication [MF67; MW67]. Moreover, extensive customer shopping histories may not be available. However, computer systems can leverage other sources of information and use them to learn customers' general shopping behavior. This knowledge can be used to identify customers with similar behavior, so the system can produce recommendations for newer customers. Note that this is just like the retailer who used experience from previous customers in dealing with new customers. Online streaming services like Netflix [BL07] or web shops like Amazon use these ideas [LSY03]. They build user behavior profiles and recommend items that "other customers also liked or bought" in the same fashion [SKR99]. Furthermore, products that fit together well or which are often "bought in a set", called suitable packages above, are recommended. The technical term for these personalized systems using big data is recommender systems [Ric+11], and they aim to support users in their decision-making processes while interacting with a large amount of information. They recommend items of interest to users based on preferences that have been expressed, either explicitly or implicitly. This means, a user could explicitly state that the "thriller movies" are their preferred genre, or the system might imply this because thriller movies have been watched in the past. Altogether, this helps to "overcome the information overload problem by exposing users to the most interesting items, and by offering novelty, surprise, and relevance" [RWZ10, RecSys'08].

Considering software engineering, the number of artifacts being dealt with is steadily increasing (cf. big data), and managing these (cf. personalizing) has become a major issue in many development projects. The most prominent example of these artifacts is source code. Hence, state-of-the-art Integrated Development Environments (IDEs) offer content-assist, code completion, and query functionality to reduce information overflow. Furthermore, ideas from the abovementioned recommender systems have recently been adapted to bolster code reuse [JHA14]. 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 [Bru12], [Ecl14a]. So, if a programmer has just created a new text object (cf. figure 1.1), the code recommender will offer a selection of methods that other programmers invoked on such objects along with their respective likelihoods.

However, source code is not the only kind of artifact created in software engineering tasks. There are also domain models, say models in general, and others. Unfortunately, compared to source code, no recommendation support for modelers is yet available [Wal13; Mic+15]. This is surprising, because some effort has been given to researching model repositories and how to preserve models [FBC06; Alt+08; LFW12; Bas+14]. As a result, there are numerous ways to put models into repositories, but relatively few ways to extract them for reuse, i.e., reutilization. If thoroughly arranged in a meta-structure, models could provide the foundation for a recommender system aimed at model reuse. For now, let us focus on (software) reuse [FK05].

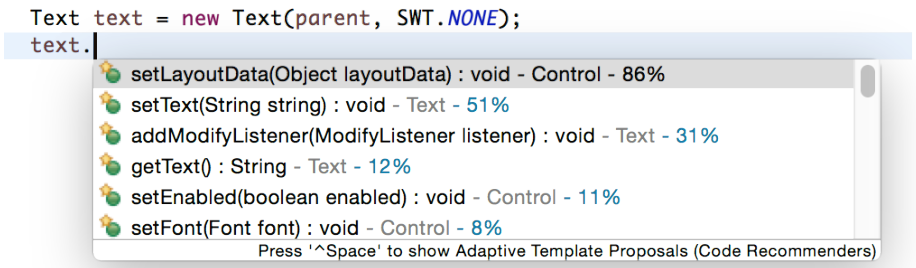


Figure 1.1.: Eclipse Code Recommenders Screenshot [Ecl14a]

In software engineering, a full life cycle of software artifacts comprises many activities. Only after the customer requirements have been elicited can design and implementation meet these needs. Hence, developers can decide whether to "make or buy" solutions. In fact, several components are known as "never ever, ever, ever code yourself", because faulty behavior would be fatal. Examples are login mechanism or date calculations [Sco13a; Sco13b]. Without going into detail, a login mechanism that can properly authorize and authenticate is considered a task that only a few teams in the world can implement flawlessly. Moreover, it may take years to construct a system that requires little or no maintenance. Similarly, date calculations are difficult because of many special cases like leap years[1], leap seconds, and countries skipping years. As a result, programmers make conscious decisions about reusing existing solutions by importing libraries based on functionality or based on their experience. Regarding models, as of today, modelers are nowhere near this, and importing a model from a library is a dream. Why is that? Are they so different?

Put another way, what are the commonalities between, on the one hand, importing a program library and reusing a fraction of its functionality, and on the other hand, considering a model library and importing or reusing a fraction of its models? They have in common that items need to be altered in the process of reutilization. Hence, function calls are often wrapped and return values are adapted; e.g., a class from the Joda-Time library called `DateTime` might offer desirable functionality, but the returned format could be incompatible with that required. Therefore, the return values need to be altered. Likewise, a model from a model library is unlikely to perfectly match a given requirement, and will need to be altered, e.g., for technical or domain reasons.

Fortunately, programmers usually have a precise idea as to what functionality is needed; thus, the question arises as to whether a modeler does, too. The answer is probably not that precisely, but this does not mean that 80% of solutions or examples cannot assist in regard of some well-cited reuse benefits for early life cycle artifacts [Cyb+98; Som11, all]: reuse is often considered a time/cost saver because of "better utilization of available resources", i.e., reinventing the wheel is not necessary. However,

[1] 29.02.2016 Airport software does not recognize leap year Google Translate https://goo.gl/EjD33N

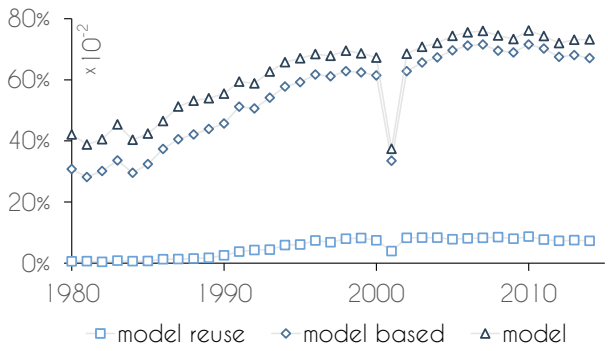


Figure 1.2.: Term Trends on SpringerLink: "Model *and X*" [Spr15]

efforts at preparing for reuse are not to be neglected, and reuse is only reasonable if the provided artifacts inherit superior quality compared to being reinvented [BB91; FI94; GAO95]. Further, systematic reuse, starting right at the beginning of the software life cycle, encourages reuse for the entire lifecycle. Additionally, it is believed that reuse improves the overall quality of software engineering artifacts, because it enables reuse of subsequent artifacts. In addition to that, and more modeling-related, reuse can unveil model elements that did not occur to the modelers, and vice versa, as consciously deleted elements state design decisions. Similarly, any renaming expresses the same intention and requires only adjustments, which are considered quicker than creation. Finally, and probably most importantly, model reuse is a step toward modeling as an "art", becoming an engineering discipline with quality standards and practices that are agreed upon and applied [Moo05], because "paradigms such as Model-Driven Development and the Model-Driven Architecture have emphasized the importance of 'good' models from the beginning of the lifecycle" [GPC05].

1.1. In Surroundings of Model Reuse

The terms "model" and "reuse" are vital for the course of this research, and so the investigation of trends in these terms is reasonable. Hence, two science-related search engines were queried with both terms. Moreover, it is desirable to combine both terms with other important terms that often occur in the course of this research, either as technologies or concepts. All of this provides a picture of the state-of-the-art in the scientific community and shows whether there is any hype around certain trends at hand. After all, it seems rather undesirable for extensive research to be chasing rainbows.

The first search engine queried to determine a trend was Google Scholar (not illustrated in the figure). The numbers peak at 1.5 million hits per year from 2000–2004 regarding "model", and decline to 200,000 in 2015. Values for "model based", which counts hits containing both terms, are marginally smaller. Regarding "reuse" and "model reuse", values peak from 2005–2009 and 2007–2011, respectively, at around 80000 and 50000,

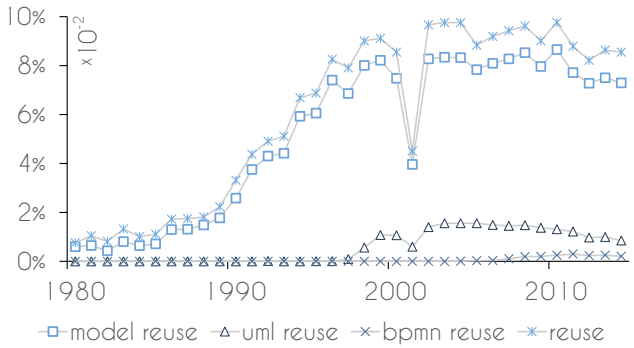


Figure 1.3.: Term Trends on SpringerLink: “Reuse *and X*” [Spr15]

with a steep decline to 40000 and 30000. These numbers demonstrate that interest in these terms faded after peaking.

The numbers mentioned above need to be taken with a grain of salt, because they show no more than the presence of words in publications. This means a term can be mentioned out of context. For example, an article can be about models, as defined later or it can be an acoustic model. Furthermore, reuse as a term is so general that a single mention can easily occur. As an example, a mathematical paper might reuse a definition or formula. In doing so, this paper is not relevant to the term we are looking for. Still, the counted numbers provide general trends, and it can be seen that all terms were used less frequently in recent years. Unfortunately, at the time of writing, Google Scholar did not allow further filtering. Hence, the decline in occurrences is across all science subjects, and numbers for computer science had to be drawn from a different source.

A possible source for analysis, which can be limited to computer science, is SpringerLink [Spr15], and figures 1.2 and 1.3 summarize the results of the noted queries. The numbers are relative to all annual publications in computer science.

The starting point for the numbers of “model” mentions in figure 1.2 is 778 or 0.4%. Thereafter, the percentage increases to 0.7% with a total of more than 55000 hits. In terms of numbers, this is a massive increase, but is only marginal relative to publications in computer science. The values for “model based” are very similar, but a slight decline can be noticed after 2010.

“Model reuse” is not as marginal as in the Google Scholar term trends, but numbers start at 11 and reach approximately 5400; this is 0.006% and 0.07%, with a peak at 0.086% in 2010. Here, “model reuse” is the linking value between figures 1.2 and 1.3. The latter shows that values and trends for “reuse” are similar to those for “model reuse”. Extra attention should be paid to the “Unified Modeling Language (UML) reuse” values. They are notable from 1997, with 16 hits, and hyped from 2005–2011, with around 750 hits. Since 2011, the values have gently declined in absolute and relative terms.

The main point is that there is no massive decline in the term trend analysis at SpringerLink, as opposed to the Google Scholar analysis. Therefore, while the terms “model” and “reuse” seem to have become less popular in some disciplines, they are still an important

(a) As Is: Copy'n'Paste

(b) To Be: Seamlessly Integrated

Figure 1.4.: Model Reuse Workflows similar to [DGL14b]

topic in computer science. Perhaps they are established as research fields, but, again, there is a need for caution when interpreting these values.

1.2. The Challenges of Model Reuse

Viewed as a development process, software engineering comprises many disciplines. Often, requirements engineering is the starting point and either deployment or retirement is the end point. In between, there are many engineering disciplines such as design, implementation, or testing, as well as supporting disciplines such as project management or configuration management.

Some engineering disciplines such as implementation can work on top of existing solutions by performing reutilization or reuse. For example, libraries are often used in the implementation stage to benefit from quality-assured solutions and reduce the development time. However, code reuse often goes way beyond this and is supported by a clever completion mechanism. For example, a programmer includes a library and the programming environment (IDE) offers relevant completion. This prevents typing errors and speeds up development.

For many development processes, regardless of whether agile or not [LA12; Kru04], models became a foundation, but reuse concerning models is still in the early stages. This is even true for the tools available to support de facto standard modeling languages like UML and Business Process Modeling Notation (BPMN). At best, a modeling environment supports modelers with type completion, so that a modeler always needs to start from scratch or conduct "copy'n'paste reuse", as shown in figure 1.4a. This is inefficient and unpleasant in several ways [DGL14b].

In fact, modeling tools are far behind state-of-the-art IDEs in terms of usability and rarely support modelers, except for model validation and automatically arranging elements [BR05; SIK15]. Consequently, if modelers seek to integrate models from a model library, they need to do so manually, as shown in figure 1.4a. The downside of this is that modelers get distracted each time they want to reuse models from a library, because they need to change context, i.e., mediate between the modeling tool and the model library. This not only makes modeling unappealing but also is likely to have a negative impact on model quality and the time spent modeling.

Hence, it appears desirable to follow a workflow similar to the abovementioned library reuse for source code, i.e., first include a model library and then get recommendations from it, as shown in figure 1.4b. This figure shows that a modeler only interacts with an IDE and can forget about the underlying model library. However, this is new ground, as we found out in an initial survey [DGL14b]. Moreover, model libraries intended for reuse are rarely researched [GL13], and evolution in this domain is not supported by such model libraries [Gan+13]. This leads to five challenges (marked ⚿ below) that we will address subsequently; these are derived and extended to what Janjic, Hummel, and Atkinson formulated for source code repositories [JHA14]. Altogether, we aim at a complete, some may say holistic, approach with a character akin to referential architectures based on operation-based model recommenders by addressing the following challenges (follow-up: pages 166, 182 and 186; contributions: section 5.4 (p. 182)):

- ⚿ *Storage Challenge:* locating and accessing.
- ⚿ *Representation Challenge:* warehousing and organizing.
- ⚿ *Harvesting Challenge:* identifying and extracting.
- ⚿ *Evolution Challenge:* changing and improving.
- ⚿ *Retrieval Challenge:* querying and retrieving.

First, the storage challenge (called the repository problem by Janjic, Hummel, and Atkinson [JHA14]) encompasses the quest for libraries that hold beneficial material. This is as true for model libraries as it is for source code repositories, because accessibility, quality issues, or copyright might not be appropriate. Note that we use the term library instead of repository, as explained later in section 3.2 (p. 48). Second, the representation problem explained by Janjic, Hummel, and Atkinson deals with organization issues and data representation for source code repositories; this is even more relevant for model libraries [JHA14; GL13]. Third, the harvesting challenge is partly specific to our domain and emerges from the necessity that model libraries hold beneficial data that can be added to the library with an acceptable or controlled degree of redundancy. Further, this challenge represents the need for harvested data to adjust to the underlying model library data structure. Fourth, the evolution challenge is specific to our domain and addresses changes of models in model libraries, especially in regard to quality. Fifth, the retrieval challenge is about how to acquire the necessary data and leverage them to get the most suitable model from a model library offered for reutilization. For a given recommender systems as an approach for solving the retrieval challenge, this is about “data processing, capturing context, producing recommendations, and presenting recommendations” [RW14].

The overall aim of this text is to provide a solution and arguments so that the following question can be countered: Will heavyweight and costly modeling be overtaken by “agile” approaches? [HM08b, Methodenkonflikt]. Discussions on model reuse are relatively rare and rudimentary, e.g., for Simulink elements [Hei12a; Hei12b], for BPMN as a requirements catalog [Mic+15], or for other early lifecycle artifacts [Cyb+98], but as of

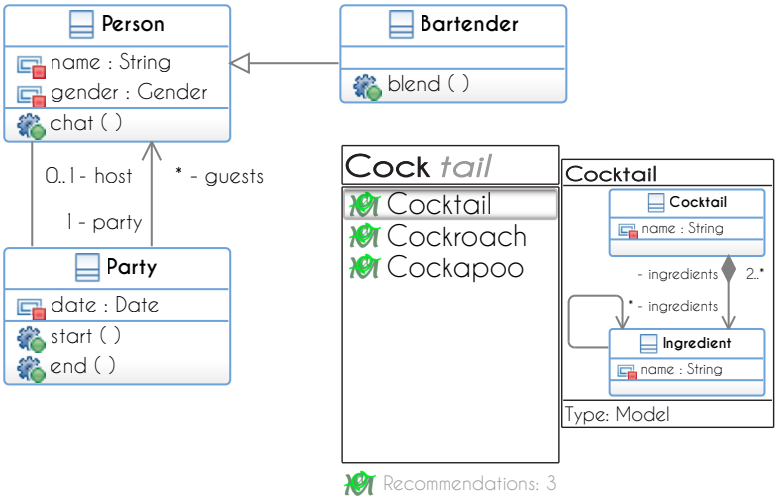


Figure 1.5.: Model Reuse Vision: Searchbox and Preview similar to [GL13]

today, there is no complete approach supporting model reuse, although it is considered more important than code reuse [Gom+04; Cyb+98; Hei12a]. However, model reuse is vital for reasons of knowledge transfer [DTC13], as well as for the abovementioned point of changing modeling into an “art”, becoming an engineering discipline with agreed and applied quality standards and practices [Moo05].

1.3. One Vision of Model Reuse

The project vision sketched in figure 1.5 was designed as a figurative drive and shows an excerpt of a class diagram editor with a search box (cf. video [Gan13b]). A user has typed in some characters, and the underlying system returned some results. These results are presented as -items in a drop-down box, and the user can step through them while previews appear to the right as class diagrams. As soon as the user picks an item, it is applied to the canvas as if the elements, i.e., classes and attributes, were created manually (cf. figure 1.6). All of this should be easy to use and seamlessly integrated in an IDE, although “the quest for simple and integrated IDEs” is not new [San78].

At first glance, the realization behind source code completion and model completion appear to be very similar, but already the data backends differ. While source code completion can rely on the grammar of the programming language, source codes analyzed from a given scope, and libraries, the data backends for model completion are mostly beneficial if a quality-assured model knowledge library is used. This should contain best practices, patterns, examples, or partial models [DGL14b]. Further, such a model knowledge library should be able to interlink all this information in an enhanced and indexed models graph library, also called a model or knowledge library [GL13].

There is no guarantee that a model, which is freshly stored in a model knowledge library, is of good quality or suits any modeler’s needs. Consequently, an approach

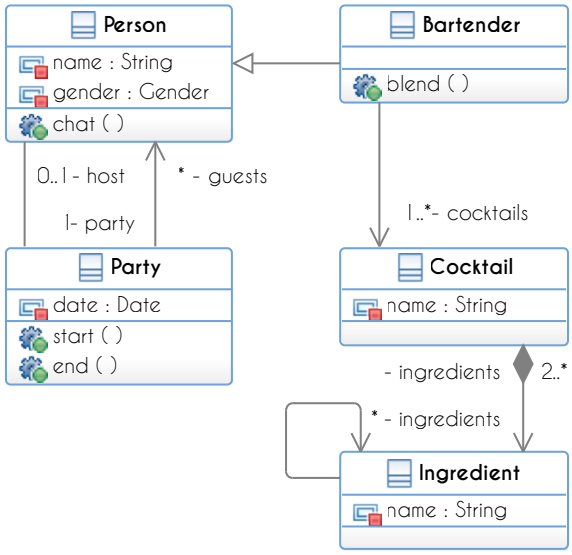


Figure 1.6.: Model Reuse Vision: Placed Recommendation similar to [GL13]

must deal with model evolution in model knowledge libraries, moving toward high-quality models. Only then can the requested and postulated "'good' models from the beginning of the lifecycle" be assured [GPC05]. Hence, the approach should track models over time and guide changes by means of a real-time quality feedback with simple quality statements [Rot+13; Gan+13]. This, hopefully, would make model reuse more attractive.

The second difference between source code and model completion lies in the retrieval mechanism, which has received relatively little attention. There is almost no research on how to find the best, i.e., fitting or matching, model for a given environment, which is sometimes called the context. The most promising approach is an adaptation of ideas from recommender systems [Jan+11; Ric+11], because model knowledge libraries tend to contain information that is not immediately apparent. An environment for experimenting with model recommenders should support retrieval experiments [DGL14a; DGL13]. These experiments could also examine how to apply the selection presented in figure 1.5 in different editors, how to use different sources of data, or how to gain sufficient information from the context, e.g., an editing history. Together, this should enable retrieval mechanisms to produce "fitting or matching" recommendations.

A third difference with respect to source code completion is the means of filling the model knowledge library or finding models for it. Enhanced source code completion can use data mining techniques and analyze existing source code repositories to fill databases, allowing the code completion to be enhanced to code recommenders [WKB09; Bru12; Ecl14a]. This means that mined data are leveraged to reorder the available completions, ordered by likelihood, as depicted in figure 1.1. Contrary to that, for models, there is no mining approach. Still, tool support can help modelers and use a harvesting mechanism for knowledge that adapts ideas from data mining and graph mining [HKP12; AW12].

The following text draws together the above mentioned issues as a motivation and

provides a sound foundation for building a solution for either classical or alternative recommender systems. Altogether, the subsequent text is based on and extends some of our previous publications [GL13; DGL13; DGL14b; DGL14a; Rot+13; Gan+13; Gan+16]. We reach conclusions in the form of the contributions in section 5.4 (p. 182) and section 6.1 (p. 186), and pick up and address the challenges outlined in section 1.2.

1.4. For Accelerated Reading and Quick Navigation

Some aspects on the style of writing and reading of scientific documents should not need further mention [CS03; CO09; SF94]; some other aspects of academic English are mentioned below [OH06]. Further, the structure of this document is a tailored version of the standard for theses, as explained later.

1.4.1. Style of this Document

This document is written to be read, and thus adheres to certain standards, mostly called text grammar, which might not be commonly known. First, this means that a writing technique called “topic sentences” is used throughout this document [OH06]. This fosters quick navigation and reading, because the first sentence of almost every paragraph provides an exact grasp of what is discussed in the following sentences. The only exceptions are paragraphs summarizing larger areas such as chapters. Second, the text is written to adhere to a linear writing style as much as possible by avoiding backward or forward references within sentences [Pic09]. At the same time, branches in argumentation are avoided as much as possible. This might appear uncommon to non-natives, but this eases reading a lot, though it does the exact opposite to the writers. Certainly, this does not mean that references throughout sections or chapters are avoided.

These references are meant to support readers as much as possible and not to be a hindrance; hence, they are designed for quick navigation as well. In detail, this means that references are provided by identifiers only if they are related to a section or within a chapter, because flipping a few pages back or forth should do the job. However, in case a reference points further, they are enhanced with a page number for convenience and quicker lookup. Certainly, literature references do not use this feature because they are gathered altogether in a designated chapter. The bibliography itself, which is close to the end of the document, provides a “Cited at page(s)” list that points to all pages on which this item was referenced. Another option for quick navigation is the reference links available in the digital format of this document. These are indicated by colored frames. Citations are framed in Cornflower blue and references to glossary terms or textual points are framed in Danube blue. Names occur only when citing related work.

The final writing aspect worth mentioning regards the gender discussion. As often as possible, we make use of the common address “we” or use plurals; sometimes, we need to address individuals as a role or as a person. In these cases, we do not mean to offend or demean any gender. If in doubt, we invite both genders to feel equally addressed.

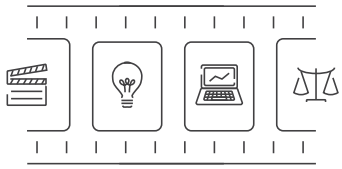


Figure 1.7.: The Document Structure

Further, we do not mean to give this document a more colloquial touch by including the reader with the somewhat casual “we”. Instead, we would like to take readers by the hand and drag them into the fascination of the course of this document. There is an interesting discussion coming up that is absolutely worth indulging: enjoy!

Regarding typefaces, we picked two fonts without serifs. The first is very similar to Helvetica and should foster convenient and pleasant reading. In fact, this font is not exactly Helvetica, but the almost identical LaTeX clone called Nimbus Sans L. The second font is similar to Courier and called `Consolas`. We use this for mono-spaced text such as data types, identifiers, or source code (cf. pseudocode 1.1).

```
1 public class WelcomeToHERMES {
2     /** The Java Comment Style */
3     public static void main(String[] arguments) {
4         // another comment
5         System.out.println("Hello HERMES! Nice to meet you :-)");
6     }
7 }
```

Pseudocode 1.1: Welcome to HERMES

Sometimes, we highlight “newly introduced content” or “technical terms” by setting it in *italics*, or emphasize that the following should be considered as `concepts` or `source code` by setting it in `typewriter`. Longer source code is set in a dedicated environment, as shown in pseudocode 1.1. In the course of this document, many definitions, symbols, and conventions recur. While we consider this easy for experienced readers, we are aware of the trouble this can cause for novices. We summarize all notation in a list of acronyms and symbols (p. 255) as well as in a glossary (p. 257) for convenience. Further, we track all major terms in an index (p. 261); a hard-copy of the acronyms, symbols, and glossary may be useful while reading.

We provide more assistance to readers in the form of guiding and summarizing figures. These are meant to provide a map and guide readers through parts of this document. Most importantly, our document architecture (cf. figure 1.7) and our project architecture (cf. figure 1.8) help to emphasize the upcoming parts in gray, as we explain in more detail below.

1.4.2. Structure of this Document

The architecture of this document comprises four main parts, as depicted in figure 1.7. The clapperboard, , represents the part used to “set the stage”, i.e., provide background

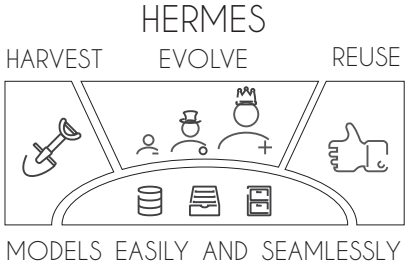


Figure 1.8.: The HERMES Project (taken from [Gan14h])

information. A light bulb, , represents the conceptual ideas, defining "operation-based model recommendations". In addition, a notebook computer, , indicates that the "HERMES" product, i.e., the realization, is introduced. Finally, the part inspecting and "assessing processes, concepts, and HERMES" in chapter 5 (p. 165) is represented by a weighing scale, . The main document closes with a part that is unmentioned in figure 1.7: looking at the road ahead while "the curtain falls". For a better overview, we present this figure at the beginning of each chapter with the upcoming part highlighted.

Further, the conceptual part could benefit from further guidance. Hence, the architecture of the HERMES project helps to structure the document in this chapter [Gan13a; Gan14g]. It comprises four parts related to "harvesting, evolving, and reusing models easily and seamlessly", as shown in figure 1.8. This figure recurs as a guide whenever a part is introduced, with the respective part highlighted. Additionally, the grid is highlighted to indicate that the foundation, i.e., "operation-based models", , are introduced.

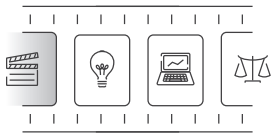
In detail, the stage is set in chapter 2 by providing an overview (section 2.1 (p. 14)) of formalities (section 2.2 (p. 14)), the conceptual environment (section 2.3 (p. 16)), use cases (subsection 2.4.1 (p. 24)), and a running example (subsection 2.4.3 (p. 26)). The recommendations in chapter 3 are built by defining "operation-based models" in section 3.1 (p. 32), explaining how to work on the "storing of models", , in section 3.2 (p. 48), how to perform the "harvesting of models", , in section 3.3 (p. 73), how to deal with "evolving models", , in section 3.4 (p. 94), and how to undertake the "reusing of models", , in section 3.5 (p. 117). The product presented in chapter 4 reiterates the project structure in section 4.1 (p. 152) and explains the respective components. First, the model data framework is introduced in section 4.2 (p. 153), and then the model mining framework is described in section 4.3 (p. 154). The model evolution framework is then explained in section 4.4 (p. 155), before the model recommender framework is introduced in section 4.5 (p. 156). The evaluation in chapter 5 provides a brief history in section 5.1 (p. 167), discusses quality assessments in terms of our development process in subsection 5.2.1 (p. 169), and considers product quality in subsection 5.2.2 (p. 172) and subsection 5.2.3 (p. 178). In addition, chapter 5 provides an assessment of our contribution to the scientific knowledge base in section 5.4 (p. 182). Finally, chapter 6 (p. 185) concludes by discussing the achieved status in section 6.1 (p. 186), outlining possible further research in section 6.2 (p. 189), and providing additional notes in section 6.3 (p. 193).

2 Setting the Stage

Don't panic!

Douglas Adams

Contents



For the course of this document, we set the research stage in the surroundings of reuse and models, provide a running example and use cases, and establish additional foundations. While we consider reuse, by and large, as a process that starts by identifying parts of a model and continues to the reutilization of models, we think of models, by and large, as representations of real-world objects formalized as UML class diagrams. Our running example is in the notation of such a UML class diagram, and a UML use case diagram provides an overview of the requirements we should meet. Finally, and with regard to additional foundations, we, like any other research, base our work on conceptual and technological approaches for which we now provide an overview and later describe in detail.

In regard to these conceptual foundations, the research environment for the operation-based model recommenders involves several disciplines in computer science. All of them are applied in an environment that treats models as first-class citizens to, eventually, achieve model reuse. Hence, we need to find (harvest) reusable parts, ensure their quality (evolve), and make them available for reutilization (reuse). All of this requires data organization in the form of a knowledge library. First, we harvest models, which means we need to identify beneficial parts of models. The relevant domain in computer science is mostly data mining [HKP12], and as our models usually inherit a graph structure, we specifically consider graph mining [AW12]. Second, we evolve models, i.e., support modelers by enhancing them. This involves software quality [Wag13] as well as software evolution [MD08]. Third, we reuse models, which requires the identification of models for reutilizations. This involves ideas from machine learning and related approaches [Mur12; Bis06]. Most notably, these are recommender systems [Jan+11; Ric+11]. To aid this, we look at context management [Bet+10].

On a technological basis, the research environment for operation-based model recommenders involves manifold technologies. First, the primary programming language is Java [Gos+13], which limits the choices for IDEs. We picked the Eclipse platform as an IDE and deployment platform [Vog13], allowing us to benefit from OSGi [OSG14], the plug-in development environment (PDE), and the Eclipse Modeling Framework (EMF) [Ste+08], which allows persistence extensions with NoSQL databases [SF13; RWE13].

2.1. Scenic Overview

Before we get into details of formalities and terms, we look at the approach at hand from a more general perspective. The approach we are about to introduce is called “Operation-Based Model Recommenders”, and considerable thought went into this title. The first thing to note is that we talk about “recommenders” and not “recommender systems”. The latter is a well-established term and some realizations, which we enable by means of our approach, can be classified as recommender systems. However, we intend to denote the entirety of possible realizations, and can only assert that these produce suggestions that are not always distinguishable from recommendations. Hence, we prefer the term recommenders as a summarizing concept and because it is established for “Code Recommenders” [Ecl14a]. Consequently, our approach could be termed “Model Recommenders”, because our primary items of consideration are “Models”, as mentioned in the title. This primarily puts our focus on models as used in software engineering, and does not neglect their structural nature as graphs. Hence, there are some similarities to graph-based recommender systems, and we explain how to convert them into traditional recommender systems by investigating “operation-based models”. This means that we quite often consider models from an operation sequences point of view, both while analyzing an editing sequence or reutilizing a model. Overall, we go into the necessary detail for the relevant formalities, concepts, requirements, and target environment.

2.2. Scenic Formalities and Conventions

For the sake of comprehensibility and simplicity, we often employ a semi-formal notation for concepts. They are rooted in mathematics and many of them relate to set and graph theory, as well as complex analysis. Hence, we reiterate some basic notation [Ros12].

We take the mathematical concept of a set, so we introduce it by providing a definition and omit examples or derived concepts such as the empty set ($\varnothing$), subsets ($\subset$), and cardinality ($|\cdot|$):

> A *set* is an unordered collection of objects, called elements or members of the set. A set is said to contain its elements. We write $a \in A$ to denote that *a* is an element of set *A*. The notation $a \notin A$ denotes that a is not an element of set A. [Ros12]

When discussing an element of a set, we often refer to it as ε and use an index for further information. For example, we often talk about classes and use the index “C” for clarification: ε_C. In other cases, we change the font and denote a special `Models` with an index “$\mathbb{M}$”: $\varepsilon_{\mathbb{M}}$. On top of sets, we use Cartesian products made of more than two sets.

> The *Cartesian product* of the sets $A_1, A_2, \ldots, A_n$, denoted by $A_1 \times A_2 \times \ldots \times A_n$, is the set of ordered *n*-tuples $(a_1, a_2, \ldots, a_n)$, where a_i belongs to A_i for $i = 1, 2, \ldots, n$. In other words, $A_1 \times A_2 \times \ldots \times A_n = \{(a_1, a_2, \ldots, a_n) | a_i \in A_i \text{ for } i = 1, 2, \ldots, n\}$. [Ros12]

Sometimes, we access individual elements of tuples and employ a dot notation instead of using a labeling function. For example, consider an element as introduced above, ε_{C}, as a tuple. If we want to examine the second element of this tuple, we simply write $\varepsilon_{\mathsf{C}}.2$, or address it as `name` to refer to the specific element.

Further, set cardinality and intersection play an occasional role. For sets, we say their cardinality is the number of elements contained within, as usual, and for tuples, we can define their cardinality as necessary, e.g., considering certain elements only. If we have a 5-tuple in which each element is a set, we can define the cardinality of this tuple as the cardinality of the second plus the last set (e.g., equation (3.51) (p. 81)). With respect to set intersection ($\cap$), we take the usual definition for sets and extend it to tuples if necessary. As an example, we can take two 5-tuples in which each element is a set and define their intersection as the pairwise intersection of the respective sets (cf. equation (3.115) (p. 133)). In addition, we use graphs as representations of models (cf. pseudocode 3.2 (p. 77)):

> A *graph* $\mathsf{G} = (\mathsf{V}, \mathsf{E})$ consists of V, a nonempty set of *vertices* (or nodes), and E, a set of *edges*. Each edge has either one or two associated vertices. [Ros12]

However, models are rarely undirected graphs. Rather, they are directed graphs, or digraphs, as we show later in figure 2.4. Further, we can alter graphs with labeling functions of different forms. These can add information to vertices or edges to form property or weighted graphs. For all graph types, we can introduce the concepts of adjacency and neighborhood:

> Two vertices u and v in an undirected graph G are called *adjacent* (or *neighbors*) in G if u and v are endpoints of an edge e of G. [Ros12]

We will need to adjust this idea of neighborhood later for two reasons. First, because models expose some unexpected graph structures when only graphical notations are considered. For example, two related classes in a class diagram are not neighbors, as we show in figure 2.4. Second, semantics requires us to change terminology to denote neighbors as successors ($\mathrm{succ}(\cdot)$). Finally, we need the degree of a vertex:

> The *degree* of a vertex in an undirected graph is the number of edges incident with it, except where a loop at a vertex contributes twice to the degree of that vertex. The degree of the vertex v is denoted by $\deg(\mathsf{v})$. [Ros12]

Again, we need to consider directed graphs and find not only the degree of a vertex, but also its in-degree and out-degree. The former denotes the edges directed to the vertex and the latter the edges directed to neighbors. In addition to graphs, and without further introduction, we sometimes use an alternative, but equivalent, representation called an adjacency matrix [Ros12]. We use these because an algorithm can access nodes directly without searching vertex sets, which enhances performance and comprehensibility.

Next to these structural formalities, we need groundwork for the operations that eventually make up our approach. We postpone a more detailed introduction, but can already

consider them as similar to functions and relations in mathematics. This means they either map or relate elements from a domain to a co-domain. Further, we sometimes design the domain and co-domain to be equal (subsection 3.5.2 (p. 118)), so we can use concatenation as with functions ($\circ$). In other cases, we take a shortcut and do not define the domain and co-domain equally for concatenation for the sake of comprehensibility (subsection 3.1.2 (p. 37)). Still, the semantics are the same and the order of reading or applying the operations is from right to left:

> Let g be a function from set A to set B and let f be a function from set B to set C. The *composition* of the functions f and g, denoted for all $a \in A$ by $f \circ g$, is defined by $(f \circ g)(a) = f(g(a))$. [Ros12]

Regarding symbols, we use different typefaces for distinction. First, calligraphic faces denote related universes. For example, a model such as m is part of the universe of all models, denoted $\mathcal{M}$. The exception is the universe of all operation-based models ($\mathcal{M}^{\mathfrak{Z}}$), which we comment on shortly. Second, fraktur faces denote things worth mentioning that are beyond the current scope. As an example, we enclose a set of rules in a definition as $\mathfrak{R}$ for the sake of completeness. Third, lowercase Greek typefaces represent elementary operations, e.g., for creating an element (π_c) or generating model recommendation candidates (ϱ_{gen}). Fourth, we use uppercase Greek typefaces for larger operations, e.g., querying (Φ). Finally, we introduce some new symbols, e.g., for an operation-based model ($\mathfrak{Z}$) or model recommendations production sequences ($\mathfrak{MP}$).

2.3. Conceptual Environment

In section 2.1, we gave a brief overview of our approach. More precisely, it is formed of, but not limited to, class-diagram-type models and touches on topics like repositories, data mining, evolution, and recommender systems. Moreover, Eclipse is the technological foundation for the realization. Therefore, we introduce related ideas and provide the necessary groundwork for subsequent concepts. First, however, we introduce our understanding of operations, which is derived from [III10, IEEE 1320.2-1998]:

> An *operation* is a mapping from the cross-product of instances of classes, and similarly arguments, to a cross-product of instances of classes.

The idea behind these operations is the methods in object orientation, because the basis of our approach will be methods that exist in the Meta Object Facility (MOF) reflective Application Programming Interface (API) (cf. subsection 3.1.2 (p. 37)) [Obj14]. However, our approach needs sequences of operations, which represent successive editing, as we introduce in section 3.1 (p. 32). Given that idea, we can look into the term models.

2.3.1. Models and Persistence

Our overview in section 2.1 mentioned that the objects under consideration are models and that we mean to produce recommendations based on some kind of persistence.

Therefore, we provide an understanding of the term model and examine which kinds we subsequently deal with as well as the types of persistence we take into account.
Models: Models, or better yet their concepts, have been around for quite some time [Lud03; Zar+14], and we do not mean to go all the way back to their roots (Latin “modulus”), but start with a more recent and common understanding by a German poet:

> B. Brecht: Der Mensch macht sich von den Dingen, mit denen er in Berührung kommt und auskommen muss, Bilder, kleine Modelle, die ihm verraten, wie sie funktionieren. [Bre74]

Roughly translated, Brecht states that humans create images, i.e., small models, from objects they interact with or need to deal with for the purpose of unveiling how these objects behave/perform. This statement underlines how the expression model has an intuitive, almost inherent, meaning, which is certainly not scientific. This might be due to its long heritage and etymology, but this is beyond the scope of this discussion. Essentially, researchers in many domains have tried to define model, resulting in myriad disagreements. This is not only true between disciplines, but also within single disciplines [Mul+12]. As an example, a discussion on the expressions model and metamodel in the field of computer science resulted in a public discussion which comprises remarks from others and clarifications by the author [Küh06b; Hes06b; Küh06a].

We do not mean to inject another disturbance to this discussion, but wrap up the current understanding so that we can work with this term. The term “model” as used in computer science is often traced back to Stachowiak [Sta73], Ludewig [Lud03], and Muller et al. [Mul+12]. Stachowiak’s understanding as a philosopher looks at models in regard of several features, denoted mapping, reduction, and pragmatic features. In the context of engineering models in computer science, these terms are altered and supplemented by Selic [Sel03]. Hence, a model must be capable of abstraction, understandable, accurate, predictive, and inexpensive. Without providing exact definitions, we can learn from this example how perspectives on models intervene with their definition. Often, distinctions are drawn for each by introducing new kinds of models, which are both prescriptive and descriptive [Mul+12; HM08b]. The agreement among them for Model-Driven Engineering (MDE) lies in their means of engineering, i.e., how they are built in terms of “representation of”, “element of”, and “conforms to” relationships, which can be summarized by mega-patterns [Jea05c; Jea05b; Jea05a]. However, models should not be confused with just another abstraction level or meta-layer [MBC09; Obj11a], although these are considerable components.

Altogether, we have that a model is a representation of either a real-world object or an object to be, and comes in incarnations of different dimensions, namely, purpose, formality, and granularity. For our applications, this maps to an understanding that a model is a “related collection of instances of meta-objects, representing (describing or prescribing) an information system or parts thereof, such as a software product” [III10, ISO/IEC 15474-1:2002]. Note that we deliberately omit the meta-level for now and consider it no more than a model of a model, i.e., an abstraction or generalization. Further, we need to breathe life into these dimensions. First, the purpose of our models is conceptual with the goal of code generation for static parts of systems, i.e., data models.

Second, this implies a high degree of correctness and formality, so a code generator can derive source code from a given model. However, this also requires an appropriate degree of granularity, because a model needs to grasp as much of a domain as possible so that the generated source code requires few adjustments.

The models that we are working on can be seen in different early phases of the software development cycle. Hence, they can be any structural model. Still, the focus is clearly on the design and development phase. This puts emphasis on domain and conceptual models. The former are the result of domain analysis and the latter explain and comprise concepts relevant to engineering projects. As both emerge in the early phases of software development, a remark on their relationship to specifications is probably considerate. We are aware of the advantages and limitations, but omit a more distinguished discussion about models and their relationship to specifications [Hes06a; HM08b].

As "UML is the de facto [standard] modeling language used by software developers during the initial stages of software development" [SA13], we pick Essential MOF (EMOF) to formalize models in domain analysis or conceptual modeling. We use EMF core metamodel (Ecore) as the implementation for our realization in section 3.1 (p. 32), because tool support is well established and the underlying frameworks have been shown to be mature in industrial projects [Vog13; Ste+08]. This also allows a change in perspective or treatment for models from either a set, operation, or graph point of view. We briefly explain models as graphs in subsection 2.4.3 and introduce the set perspective and operation-based model in greater detail in section 3.1 (p. 32). Thereafter, we make use of a perspective or treatment as needed. While harvesting models deals with the status of a model and treats it as a graph, model evolution and reuse come back to operation-based models. Note that this also changes our understanding of models to be an Ecore model most of the time, which can often be considered a class diagram type model.

Persistence: For the sake of simplicity, let us disagree with the statement that "everything is a model" for a moment [Béz04]. Instead, let us look at just UML models and how they are organized. There are all-in-one models containing everything, or we can separate them into files, e.g., XML Metadata Interchange (XMI). Further, we can organize file structures. Alternatively, we can put models in databases as binary large objects (BLOB) to replace the file structure or dissolve them in graph databases [BK14]. This makes models remotely available, and we distinguish these options using the SEVocab [III10].

One of the most-used terms for the storage of software-related artifacts is repository. Often, these are used for configuration management purposes, and hence provide version control mechanisms, and some have a dedicated organization. As a further distinction, repositories can be subdivided into, e.g., data, integrated, master, and software repositories. This means they can purely provide a storage function, storing all available information such as models, tools, and measurements, be the master copy of everything, or simply provide a permanent archival of software and respective artifacts. Closely related to this idea of software repositories are software libraries, which have an additional purpose of aiding in software development or maintenance.

Later, in section 3.2 (p. 48), we will derive the term knowledge library, which picks up

this “aiding” aspect of software libraries and combines it with aspects of a knowledge base. The latter is often considered a collection of expertise in a domain with additional inference rules. As we only consider the expertise and put the knowledge extraction elsewhere, we build the abovementioned term.

2.3.2. Promising and Supporting Approaches

The abovementioned storage for software-related artifacts, regardless of whether it eventually forms a knowledge library, must be filled with data for reutilizations. Both aspects can build on previous research relevant to these issues.

Data and Graph Mining: Data mining is often considered to be data analysis for the purpose of finding systematic relationships and transferring them to new data [Ama+11; HKP12; Men14]. Note that we avoid talking about “consistent patterns”, as usually considered in data mining, because patterns have a different meaning in our software engineering environment. Regardless, data mining studies datasets using techniques from machine learning and other fields, so extracted information is more comprehensible. The classical application area is “big data”, which refers to huge datasets that are considered difficult to analyze manually (hence the term knowledge discovery and the saying “separate gold from the rocks” to mean finding something beneficial).

For our application, the data at hand are not necessarily “big data”, but we still aim to extract information from a potential mess. Thus, techniques from data mining and machine learning are similarly applicable. As an example, consider a huge model and the task of extracting beneficial and unknown parts as potentially reusable components. This involves finding the known parts (in a library) and identifying reusable parts, areas that have been extensively studied in machine learning and data mining. However, we must keep in mind that our models inherit a certain structure, as discussed above (also cf. figure 2.4). In some respects, this makes data mining more like graph data mining.

Hence, one central consideration is clustering, which is the task of grouping and structuring data without taking into account known structures. In data mining, machine learning algorithms such as k-means clustering or community detection are popular [Ama+11; AW12]. The former groups a set of data into k partitions with respect to their features. These can be coordinates and the partitions can be circles in a coordinate system. The latter tries to find a subset of nodes from a graph such that all nodes in the subset are strongly connected. We introduce clustering algorithms in subsection 3.3.2 (p. 75), some of which leverage textual information available in models to give a form of text mining.

With regard to text mining, we apply and base our ideas on two concepts. First, the term frequency-inverse document frequency (tf-idf) realizes the idea of grasping the importance of a term in a document. It does so by relating the term occurrence relative to other term occurrences and documents [Men14]. However, tf-idf is only reasonable after removing certain words of low entropy, e.g., articles. Hence, we often apply stemming and stop word removal implicitly without any mention of it [MRS08; Wil06; Por80].

Further, the data mining task, which we call harvesting, retains our knowledge library regarding both its data and its structure. Therefore, neighborhood relationships from

data mining, e.g., the k-nearest neighbors (kNN) [Men14], obtain a different meaning, as introduced for graphs in section 2.2 and as shown in section 3.3 (p. 73).

Recommender Systems: Closely related to data mining and machine learning, recommender systems come into the picture [Ric+11; Jan+11]. They are information filtering systems that are rooted in decision support [Spr80], but they became popular in recent years for web systems and shops. In these environments, a recommender system tries to get the most valuable information out of a mass of information for a customer. The most popular examples at the time of writing are Amazon and Netflix [LSY03; BL07]. Both use recommender systems to calculate product or movie recommendations based on user profiles. In more detail, they leverage their experience on what a user liked or disliked to predict a preference based on how similar users behaved in the past.

Recommender systems produce result lists with so-called (recommendation) items, and the respective algorithms are commonly subdivided into four types [RRS11]. First, the abovementioned recommender system is called collaborative-filtering and is considered the most popular and widely used approach among recommender systems. It recommends items that other users of similar taste liked. Hence, user ratings and similarity between users are cornerstones. Second, content-based recommender systems take the similarity between items into account in producing recommendations. They use user tastes as well as meta-data, often called features, for comparing items. This means that a learning phase with respect to the features is necessary. Hence, content analysis and item meta-data are cornerstones [LGS11]. Third, knowledge-based recommender systems take domain knowledge into account and derive a degree of user need or usefulness. Thus, some metric regarding user needs, which is called the “problem description”, must be matched by recommendations. These may come from a collection of domain knowledge, like a knowledge library. Fourth, community-based recommender systems work on the basis of “tell me who your friends are, and I will tell you who you are”. The idea behind this is that recommendations from friends are taken more seriously. They are also called social recommender systems, because they are mostly applied in social networks. Each of these types has its individual drawbacks. Hence, issues like the cold start problem, when no information about a user is available, are approached by hybrid recommender systems that combine the above methods.

Shifting the application domain from web systems and shops to software engineering tasks changes certain considerations. The majority of research in this field is conducted for source code [Bru12; Hei12a; Wal13; RW14; JHA14], but program transformation, bug prediction, and assignment recommendation are covered as well [BR14]. By and large, the issues addressed so far concern information overload and require further research in respect of the storage, representation, and retrieval problem, as mentioned in section 1.2 (p. 6) [JHA14].

2.3.3. Software and Model Reuse

It is human nature not to reinvent the wheel and to reuse existing solutions in everyday life [LL10]. This is also true for computer science and, more precisely, software engineering.

Here, a common distinction on reuse is made between software reuse and concept reuse [Som11]. Whereas reuse in general concerns any artifacts of software development, the more precise software reuse refers to project artifacts, and the more abstract concept reuse refers to project-independent artifacts. As a very simple example, a source code run multiple times is already software reuse, and it does not matter if we think of a loop, method, object, class, component, or subsystem. In contrast, concept reuse looks at a broader perspective and we can think of design patterns [Gam+95], best practices [LR07], or reference architectures [TDM10]. All of these are independent of a particular project and need some effort for application, but provide benefits as well. Note that we must not confuse this understanding of a concept with that of conceptual modeling, which refers to terms for domain objects. Hence, we will deal with concept reuse in the context of conceptual modeling and only marginally in the understanding that contrasts with software reuse.

To start, we employ some rather general knowledge, rather than scientific understanding of software reuse, and refine it throughout the discussion.

> *Software reuse* refers to using existing software artifacts during the construction of a new software system. [Kru92]

The important point here is the continuous nature, i.e., an inherent process, and the use of existing parts in new “wholes”. Other than that, many concerns, e.g., alteration, are unmentioned. Subsequently, we do not mean to provide an in-depth introduction to software reuse [Bal01; Som11; LL10], but build up to model reuse. An elaborate discourse on software reuse is provided by Mili et al. [Mil+02].

Software Reuse: Turning to the attention paid to software reuse, we learn that design phases in other engineering disciplines encompass a dedicated activity for reuse, i.e., build the new products using as many existing parts as reasonable [Som11]. For software engineering development processes, no such design activity is mentioned in the mainstream. Instead, the all-anew paradigm is predominant [Som11]. Further, a clear indicator of the attention paid to software reuse is the fact that the respective scientific conference (ICSR - International Conference on Software Reuse) is only biannual. An even stronger indicator comes from chapter 1 of an introduction to IEEE Standard 1517 “Reuse Processes” for Software Life Cycle Processes [IEE10]—a chapter named “Reuse is Boring!” [McC01]. After a brief definition, it starts as follows: “Although a potentially powerful technology, reuse has never been counted among the most interesting software topics. In truth, most software professionals consider reuse downright boring.” So, is software reuse deemed to fail? Or is it not of interest, because it has been solved [Pou99]?

First envisioned in the late 1960s for component libraries [McI68], software reuse proved successful in terms of productivity and quality in America during the 1970s and in Japan in the 1980s ([LP79] according to [Pri91], [Mat84]). These success stories concern source code reuse of several kinds, and the first essay discussing these appeared in the early 1980s [Sta84]. The discussion of reuse activities is streamlined into five problem areas: information retrieval, software generators, component composition paradigm, program understanding, and benefit analysis. Another observation made at the same time regards

concrete hashing implementations: “it is the abstraction and not the concrete instance that gets reutilized” [Sta84]. Another support for streamlining concerns artifact classification in terms of abstraction level, customization method, and reusability conditions [LSW87].

Along with the trend for object orientation in the early 1990s, existing software reuse approaches were assessed and classified in terms of how artifacts are “abstracted, selected, specialized, and integrated” [Kru92]. This puts more emphasis on software reuse as a process and fosters a more precise idea, which we gain by fast forwarding to the understanding at the turn of the century:

> *Software reuse* is the process whereby an organization defines a set of systematic operating procedures to specify, produce, classify, retrieve, and adapt software artifacts for the purpose of using them in its development activities. [Mil+02]

This definition became widely accepted [Som11; LL10], and it will later serve as the foundation for our understanding. However, to our mind, one vital facet included by others is missing: “Software reuse is the use of [...] software knowledge to construct new software” [FK05]. Note that knowledge also comprises tacit parts, which surpasses the common understanding of software artifacts. This is due to the focus of the abovementioned source on reuse as related to product lines, including so-called domain engineering (product line engineering), which we consider as closely related to is meant by software knowledge.

Returning to software libraries [MMM98], another perspective on software knowledge is as the “wisdom of the crowds” [Bru12]. Here, source code from public libraries, which are more precisely repositories in this case, is analyzed by means of recommender systems, and hence, the reuse process is automated in several ways. The goal is to manage the information overload problem. Similarly, an approach for software libraries employs ideas of recommender systems [Hei12a]. Therefore, we can conclude that the information retrieval problem mentioned above has recently been approached in a different way to traditional methods.

This contrasts with mostly planned software reuse, as defined for an initially introduced process, but there is another, unplanned side of the coin. This kind of software reuse goes by many names, e.g., code scavenging, ad hoc, opportunistic, copy-and-paste, or pragmatic reuse [HW12, for respective references], and it is often a reality in industrial projects. The fundamental difference from other source code reuse is that “the functionality itself is a legitimate target for modification” [HW12]. This will also need to hold true for our model reuse.

Model Reuse: The models we are dealing with are often results of domain analysis, and it was stated in the 1980s that “reuse of domain analysis information [...] is the most powerful brand of reuse” [Nei84; PFW95; Cyb+98]. Only five parts of a related domain description need updating. In our case, transformations are not source-to-source, but model-to-text transformations.

Further, we have already learned that reuse can be planned or unplanned, and that the latter is preferable in our case. Still, there are some lessons to be learned from planned model reuse. An example alters the UML metamodel and extends it using so-called reuse

contracts [Luc97; MLS98]. These contracts are explicit documentation between a reuse provider, who defines how an element can be used, and a reuser, who is supposed to document how it is used and how it evolved. Certainly, this solves the issue by changing the game, i.e., change to the foundations of UML. Further, it neglects problems of retrieval and disregards experience from source code reuse, which shows that unplanned reuse is industrial reality [HW12]. Therefore, model reuse should serve a pragmatic middle ground, which is more supporting than enforcing:

> *Model reuse* is a systematic and guided process for preserving modeled knowledge, avoiding redundancy, and supporting quality assurance, as well as retrieving and reutilizing modeled knowledge.

This definition alters the definition given by Mili et al. in a few respects [Mil+02]: First, the software artifacts are restricted to modeled knowledge, which otherwise surpasses modeled artifacts. Second, the organizational perspective is omitted, although model reuse is more reasonable for cross-project application. Third, we altered the set of systematic operating procedures to a systematic and guided process, because processes in our understanding comprise activities that express similar semantics, as for software reuse, but have been adjusted to modeling and modeled knowledge. Hence, the procedures to retrieve and adapt, for example, became retrieve and reutilize. Note, that our process comprises quality assurance as well. This is attributed to the subjective nature of model quality and a potentially pragmatic nature of preservation [Moo05].

We intentionally include this perspective of pragmatic model reuse, because similar to pragmatic source code reuse, we see models as a legitimate target for modification. We think this because we aim at only so-called eighty percent solutions. Contrary to pragmatic source code reuse, our approach is intended to support localizing models for reuse [HW12]. This exceeds repository functionality (cf. [Bel99]), and we mean to do so by building our solution on a knowledge library and employing ideas from recommender systems for retrieval. Further, browsing functionality is essential, so the classification and retrieval problem introduced above can be approached in a similar manner to “faceted schema” [PF87]. In a nutshell, we will investigate pragmatic model reuse subsequently, but refer to it simply as model reuse, because we see unaltered reutilizations as the exception rather than the rule.

Altogether, the current efforts in recommender systems for software engineering tasks focus on the reuse of domain knowledge, but this excludes modeling tasks, as we showed above. Only some rather brief digressions regarding recommender systems in modeling have been conducted so far [SA13]. As an example, one approach compares collaborative filtering and association rules for rather specific modeling tasks in Simulink, but shows how modelers benefit from element recommendations, and we take this as an encouraging first demonstration [Hei12a; Hei12b].

Altogether, we do not see the (model) reuse challenge as being solved [Pou99], and agree that the library (or storing), representation, harvesting, evolution, and retrieval (or reutilizations) issues have not yet been addressed appropriately, as we explained in section 1.2 (p. 6) [JHA14]. As a note, this also addresses some known scalability issues in modeling regarding MDE [KPP09; Béz+05].

2.4. Requirements and Design

The concepts developed in the course of this research build the foundation for a software prototype. Hence, we can subsequently formulate requirements in the form of use cases and a demonstration walkthrough. Further, we derive a subsystem decomposition from the given use cases, and introduce a running example to demonstrate the concepts.

2.4.1. Top Level Use Cases

As a frame for a potential software prototype, we developed the use cases presented in the use case diagram in figure 2.1. The actual method used originated in openUMF [Hof13; HLN09], but we do not provide the full extent here for the sake of brevity. Instead, we present a more prosaic explanation, a demonstration walkthrough, and a derived subsystem decomposition. Note that all use cases should be considered with the storage of models, which we later denote as a knowledge library.

Given a model editor with a model that is currently being edited, we can start harvesting potentially beneficial parts, summarized as "Harvest Model" in figure 2.1. This use case consists of the compulsory use cases `Split Model` and `Store Model`. The former analyzes the currently edited model, proposes potentially beneficial parts, and presents them to be reviewed and edited by the `Modeler`. As soon as this is done, the latter use case represents the actual persistence. This requires the `Modeler` to provide additional meta-data and the `knowledge library` to persist the beneficial parts. An optional use case concerns `Mark Elements`, which refers to the identification of elements already available in the `knowledge library` and presentation of them to the `Modeler`. Regardless of whether this optional use case is executed in `Harvest Model`, the system leaves the status of editors opened for each beneficial part for storage in the `knowledge library` to start the evolution.

With a model opened in a dedicated evolution perspective, which displays relevant information, we can start evolving the models. All use cases summarized by "Evolve Model" are optional. Thus, `Edit Model` monitors the editing of a `Modeler` on a model and provides guidance in case of problems. As soon as the `Modeler` has finished by saving the model, a new version is produced and saved in the `knowledge library`. Independently, a `Review Model` describes how a `Modeler` is guided through an assessment of the currently opened model by focusing on a certain aspect. Further, a `Stage Model` denotes how a `Modeler` can change the quality assessment of a model in the `knowledge library`. The system leaves the `Evolve Model` in case the model editor is closed. Additionally, an `Evolve Model` can be started directly from the `knowledge library`.

While working on an unrelated model, the `Modeler` can start the use case "Reuse Model". This comprises a compulsory `Query Model` and a `Pick Model`. The `Query Model` can happen either reactively on the request of the `Modeler` or proactively by, e.g., a timeout. Note that we omit this actor in figure 2.1. In a reactive case, the `Modeler` specifies a request, so the `knowledge library` can be queried. The results from this

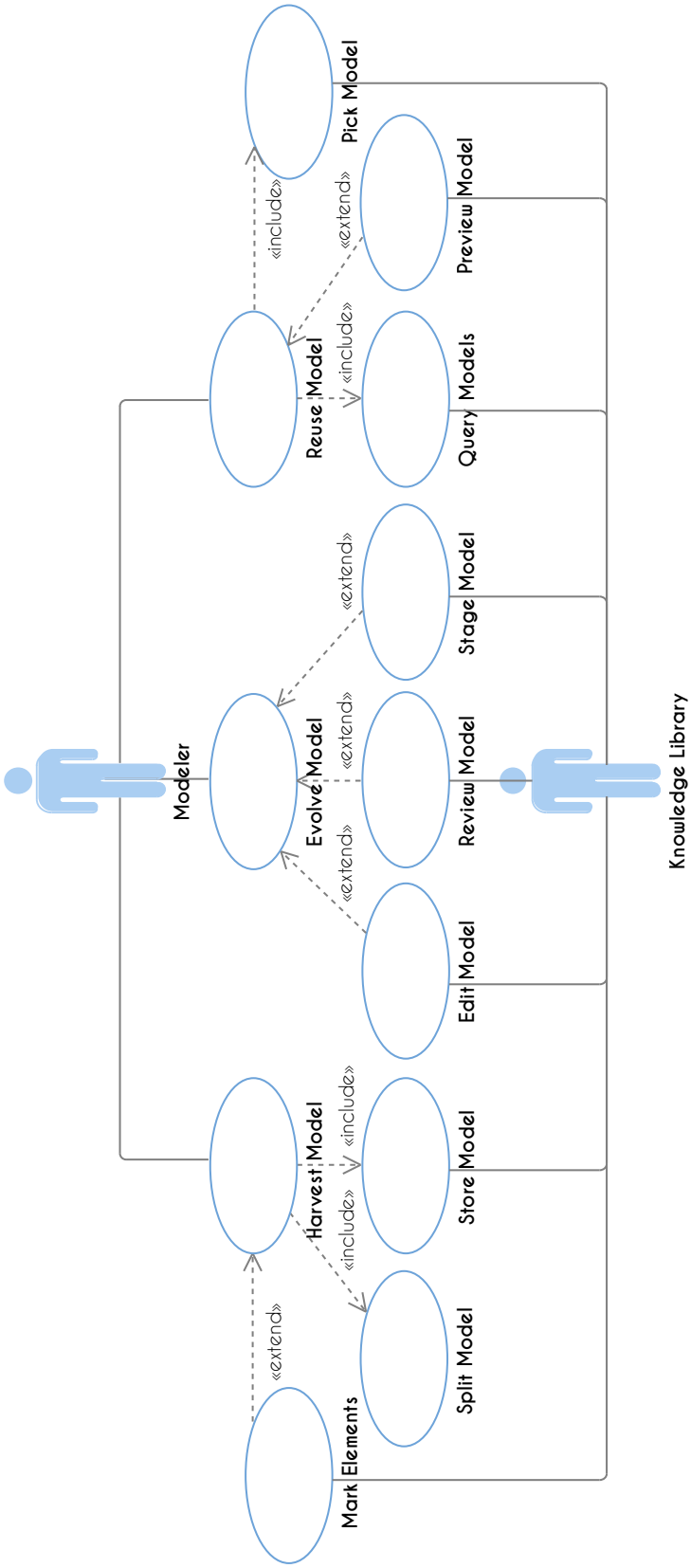


Figure 2.1.: Use Cases structured congruent to figure 1.8

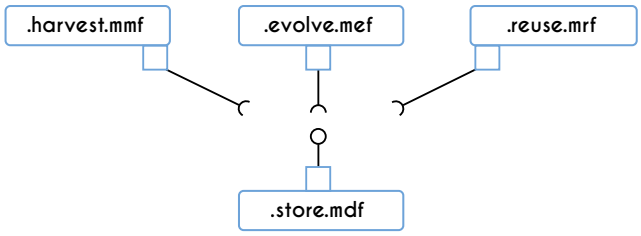


Figure 2.2.: HERMES Subsystem Decomposition

query are processed and the `Modeler` can `Pick [a] Model`. The `Modeler` may wish to `Preview Models` to find a suitable model or cancel the operation. In the latter case, the system status returns to the initial state; otherwise, the selected model is applied to the editor canvas.

Altogether, we can summarize our use cases as a demonstration walkthrough: Given a model, the showcase comprises screenshots of our software prototype, as presented in section 4.6 (p. 161). These screenshots show how known elements are marked and that submodels are found during harvesting, as illustrated in figure 4.3 (p. 154). Further, they illustrate how quality guidance is provided during evolution, as can be seen in figure 4.4 (p. 155). Finally, the screenshots depict how querying as well as previewing works during reuse, as depicted in figure 4.6 (p. 157). We have included a screenshot of our knowledge library browser in figure 4.2 (p. 153) to serve as the starting point for editing models, which actually starts the evolution.

2.4.2. Subsystem Decomposition

Given the abovementioned use cases, we can undertake a subsystem decomposition. In fact, each top-level use case from figure 2.1 corresponds to a subsystem in figure 2.2. Therefore, one will realize the harvesting (`.harvest.mmf`), one the evolution (`.evolve.mef`), one the reuse (`.reuse.mrf`), and one the knowledge library, which we later denote as store (`.store.mdf`). Hence, we postpone more detailed explanations to section 4.1 (p. 152), where we match the subsystem decomposition with the conceptual architecture.

2.4.3. Running Example

The course of the following text makes use of numerous examples, and our recurring running example is depicted in figure 2.3. This is a model concerning three areas, namely `Vehicles`, `Passengers`, and `Airports`. Each of these terms are related and enriched with further concepts. Hence, `Passengers` make up `Crew` and `Travelers`, both of which can be further refined. The same holds true for the `Airport`, which comprises several items of infrastructure such as `Hangars`, `Towers`, or `Runways`. Finally, the `Vehicles` include `Cars` or `Planes`. Note that this model does not contain compositions, which is a negligible detail, because they do not provide any benefit.

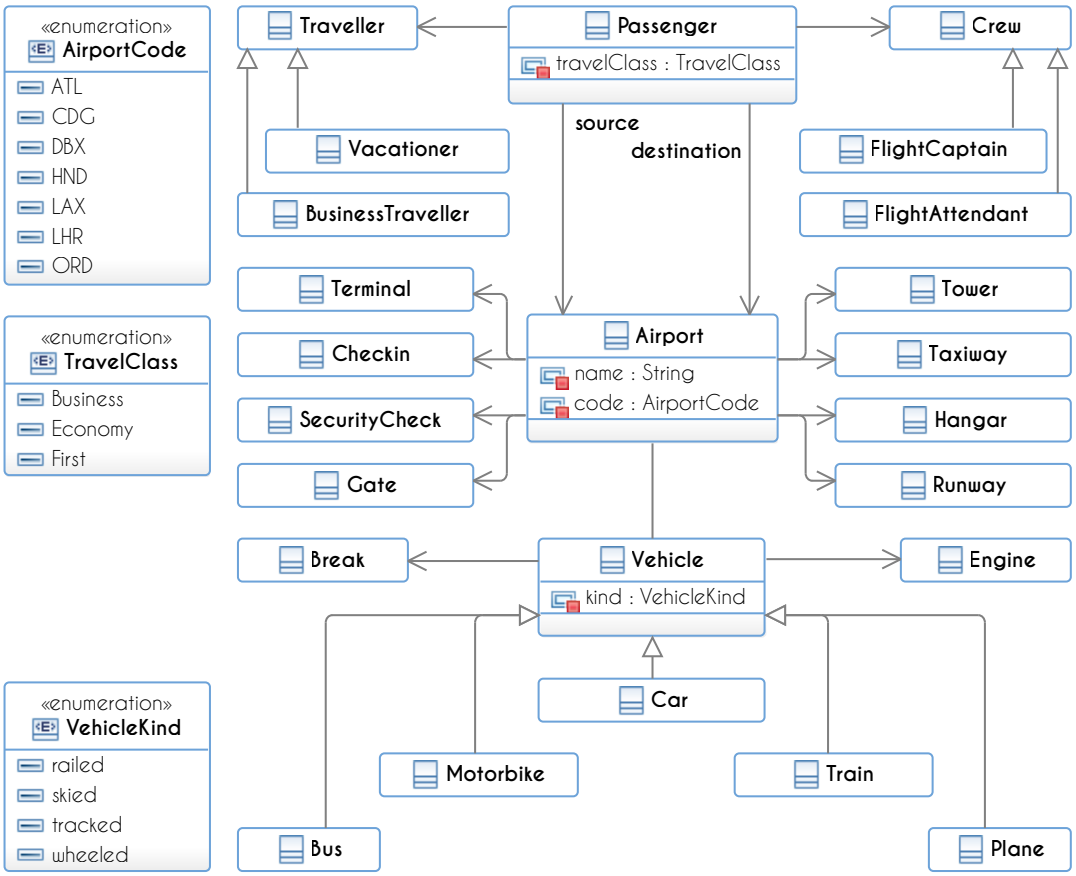
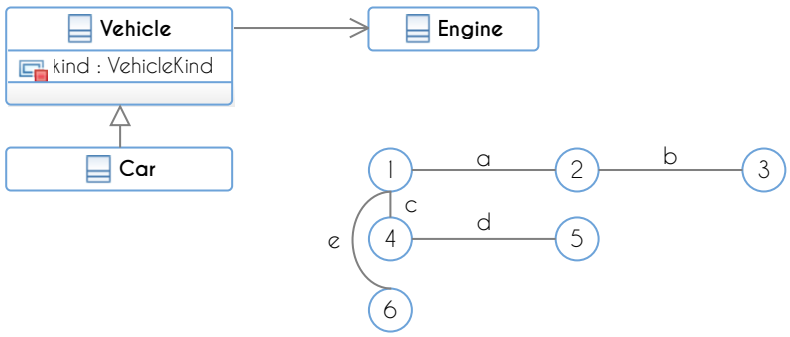


Figure 2.3.: Running Example: Airport and Surroundings

We will use this running example to demonstrate several concepts. For example, we will demonstrate how it can be written more formally as a tuple of sets or as a sequence of operations, and how it can be split up, edited, or recommended for reuse. Many of these treatments will need a slightly different perspective, but all will be rooted in either a tuple form or operation-based form, as we explain later. Hence, we will provide more details later as needed.

One detail regarding the inherent graph structure of class diagrams is worth mentioning now. Figure 2.4 shows an excerpt of figure 2.3 and its representation as a model graph. We have already used the notation for vertices and edges common in EMF, but we ignore the prefixing “E/e” because we will come back to it in section 3.1. For several reasons, the count of vertices is not obvious. First, the reference between `Vehicle` and `Engine` denoted by an edge becomes a vertex (2) in the model graph, but the generalization for the `Car` remains an edge (e). Further, the elements contained in `Vehicle` are decomposed into two vertices (4 and 5). We will return to these details of decomposition relationships and contained elements of models in graphs several times, and discuss them as required.



Vertices				Edges			
EClass	EEnum	EAttribute	EReference	eReferences	eAttributes	eType	eSuperTypes
1: Vehicle 3: Engine 6: Car	5: VehicleKind	4: kind	2:noName	a, b	c	d	e

Figure 2.4.: Vehicle Excerpt from figure 2.3 as Model Graph

2.5. Realization Environment

Demonstrating the developed concepts by implementing a software prototype requires an environment, and we opted for what was, at the time of writing, the most popular development platform available: Eclipse [Vog13]. For some parts, we added software from other, unrelated environments. In particular, this concerns the point at which we started distributing our solution.

2.5.1. Eclipse and the Eclipse Modeling Framework

At the time of writing, the Eclipse IDE was the most popular development environment and had just transitioned from series 3.x to 4.x [Vog13]. This changed some architectural cornerstones, which means that a system providing core functionality as singletons was replaced by dependency injection [Gam+95; JF88]. This inversion of control was intended to ease development, most notably for projects using Eclipse as a Rich Client Platform (Eclipse) (RCP). In such cases, core functionality like resource handling, bundle management, or Graphical User Interface (GUI) foundations are now available through annotations. Still, a compatibility layer allows 3.x development and ensures that legacy plug-ins run in the Eclipse 4.x series, as illustrated in figure 2.5. We employ Eclipse as our IDE for development and as a basis platform (RCP) for our HERMES products, i.e., the HERMES IDE for demonstration and the HERMES Software Development Kit (SDK) for sophisticated development (cf. section 4.6 (p. 161)).

Another notable aspect in figure 2.5 is that EMF Core is a fundamental, non-optional element of the Eclipse 4.x architecture. EMF is short for Eclipse Modeling Framework and is considered one of the standard tools for model-driven development [Ste+08]. It supports models in a similar way to class diagrams, which are in fact more akin to EMOF models, and enable successive code generation. To that end, we can consider EMF models as

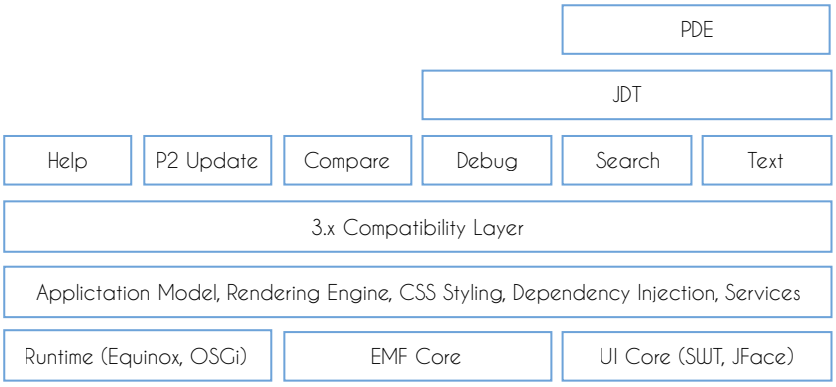


Figure 2.5.: Base Components of Eclipse 4.x RCP Applications [Vog13]

domain models and the generated prototypes as experimental software for incremental and iterative development. However, EMF Core allows reflective interpretation as well, which is why it is part of the Eclipse 4.x architecture. The so-called application models are held as EMF models that can be changed during runtime, which might change the entire program appearance. Thus, EMF models are essential in Eclipse products and are used on a daily basis in industry-style applications.

Other than that, EMF models, which we denote as models, are often called metamodels, because they represent the structure for some instance. Hence, rather abstract concepts from a model such as “Statemachine” can be instantiated as “Traffic Lights”, “Door Sensors”, and “Soft Drink Machines”. The important consideration for us is that we use the EMF description of a modeling language, i.e., Ecore, as our foundation for models. This is sufficient and does not reduce the generality, because “Essential Meta Object Facility (EMOF) [...] quite closely resembles Ecore” [Ste+08].

A note on the “meta discourse” [Jea05c; Jea05b; Jea05a]: We generally avoid it and leave readers to decide whether a model is “meta” enough or needs more abstraction. In a sense, the discussion comes down to the question of whether Ecore, i.e., the `ecore.ecore` file, is the meta-meta model and the Ecore model, i.e., the `statemachine.ecore`, is the metamodel; then, the runtime instance, i.e., the `trafficlights.xmi`, is the model. Alternatively, if Ecore (`ecore.ecore`) is the metamodel and the Ecore model (`statemachine.ecore`) is the model, then the runtime instance (`trafficlights.xmi`) is just an instance. The Object Management Group (OMG) deliberately dropped the idea of numbering meta-levels of models and modeling languages [Obj14], though the figure is still popular and we use it for classification in section 3.1 (p. 32). On the plus side for the “meta” argument are the phonetics and pretentious sophistication.

2.5.2. Other Software and Tools

Next to the Eclipse IDE and RCP, there are other pieces of software involved in either realizing or developing our prototype, because the Eclipse ecosystem and orbit do not provide everything we need. Regarding our prototype, we built our data backend for

textual files or databases, with the latter being particularly suitable for graph databases such as Neo4J and Rexster [RWE13; RN10]. Alternatively, the Hibernate ORM served a MySQL database. For querying purposes, we used either the functionality provided by MySQL or, for the graph databases, Apache Lucene and elasticsearch [GT15]. In addition, files in the data backend are under git version control [CS09; LM12].

Regarding our development, there were additional tools, frameworks, and libraries involved, and we provide a short list of the essential ones grouped according to their purpose. Almost the entire prototype is written in Java [Gos+13], and this makes Maven a strong candidate for our build environment. In fact, we use Maven 3 adjusted for Eclipse, which is called Maven Tycho. This enables continuous integration with a Hudson or Jenkins server, which is bound to a SonarQube server for quality assurance in terms of source code metrics and test coverage. The tests themselves are mostly of Hamcrest style, which is an adapted Junit with fluent interfaces. Further, mockito was used for mocking and stubbing. The GUI tests were based on SWTBot, which also served as a screen-shot engine for documentation. In addition to the logging provided by Maven Tycho and Hudson, we implemented our own logging framework built on SLF4J with LOGBack as a backend and Beagle as a viewer.

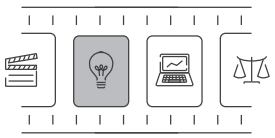
As this environment is rather complex, we simplified the development by virtualization. This allowed a one-script setup and, hence, a local git clone could start an entire build. The clone would install a Vagrant and Docker environment, and setup Docker images for a Jenkins, Nexus, SonarQube, MySQL, Rexster, and elasticsearch server. After that, the actual build would run in the Jenkins server. All that is necessary for this setup is a VirtualBox and Maven 3 installation.

3 Operation-Based Model Recommendations

In my righteous own mind
I adore and preach the insanity
you gave to me.

ANDERS FRIDÉN

Contents

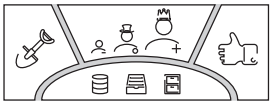


Any course of actions in a systematic process builds on a conceptual foundation. This is as true for business workflows as it is for systematic reuse. The former might be a simple workflow in which an employee registers for a holiday with the underlying concepts of organizational roles and tasks, which eventually leads to approved or rejected holidays. The latter, systematic reuse [IEE10], also comprises involved roles and tasks, as we subsequently examine, tailored to model reuse and addressing the issues described in more detail in our mission statement in section 1.2 (p. 6) and refined in subsection 2.3.3 (p. 20).

Hence, we lay the groundwork for our model reuse approach by establishing two perspectives on models. First, models as "white-boxes", which we view as graphs or sequences of editing operations (cf. subsection 2.3.1 (p. 16) and cf. section 3.1), and second, models as "gray-boxes", which concerns how to organize and interlink them in a model knowledge library (cf. section 3.2). Then, we take this groundwork and walk through the three tasks of the model reuse process. First, we identify parts of models and put them away for reuse (section 3.3). Second, we define our model evolution approach for model knowledge libraries (section 3.4). Third, we describe our model reutilization approach (section 3.5). Mind that we employ our running example "Airport and Surroundings" from subsection 2.4.3 (p. 26) to demonstrate the tasks at hand. Hence, we show how the `Airport`, `Passenger`, and `Vehicle` parts can form a model knowledge library, i.e., how they are stored; how the central `Airport` parts can be identified as beneficial for a model knowledge library, i.e., how they are harvested; how the `Airport` model can be enhanced to make them more reusable, i.e., how they are evolved; and how the `Airport` model can be retrieved and reutilized, i.e., how they are reused.

Altogether, we formulate an approach for model reuse that is complete and supports the process of systematic model reuse in its entirety. This comprises semi-formal foundations and concepts that support the storing, harvesting, evolution, and reuse of models. Further, we can leverage these foundations to feed properties to widely available recommender systems, but we go beyond that and provide an additional and extended approach.

3.1. Operation-Based Models



In subsection 2.3.1 (p. 16), we introduced models and followed an approach that treats them as directed graphs, or digraphs. These graphs can be considered as linked trees, which enable intuitive memory and persistence structures. While the former eases the editing process, the latter are intended for long-term storage in XMI documents [Ste+08].

However, models are rarely treated as being carved in stone, because iterative and incremental developments require continuous alteration. This can be rooted in different causes, which are mainly of syntactic or semantic nature. The syntactic nature mostly deals with restructurings, which are often called refactorings, although an actual refactoring requires some subsequent validation to be applied [Fow99]. In contrast, the semantic nature realizes new, changed, or dropped requirements on models.

Together, these alterations lead to editing-support functionality of several kinds. As a first example, error preserving functionality was sought. This means, e.g., that a checker was required to ensure that every attribute has a valid type or that no duplicate identifiers are defined. As a second example, transformation functionality could be required, so that a model can be used in different environments. For example, a workflow model could be transformed into a Petri net so that it can be executed by an engine. As a final example, a modeling environment could require collaborative editing functionality. This needs asynchronous transactions on a model to retain a valid state. As these examples differ, their formal foundations do as well.

Hence, many attempts at formalizing model alterations were undertaken for these different purposes. The most intuitive way to formalize model alterations for most computer scientists and mathematicians is derived from set theory [AS08; MBC09; KE14]. This formalization is often chosen for reasoning about types and related topics. Focusing on the structural nature of many models, formalizations based on graph theory are popular for reasoning about changes and structures [Her11]. Combining this with a membership equational logic, and treating models as directed graphs, yields formalizations targeted at high-performance processing and nondeterministic concurrent computations [Rom+07]. Yet another approach for formalizing model alterations with graph theory is aimed at transformations. This solves problems regarding lost updates and bidirectional processing. In more detail, transformations are formalized by triple graph grammars (TGGs) [Sch95]. A further approach enhances formalization from set theory to an algebra focusing on collaborative editing, i.e., conjunctive, disjunctive, and negation properties of operations [Kög11]. Moreover, models can be formalized by operations similar to command objects [BG93]. Starting from an empty element, a chain of operations describes the resulting state of a model. We mainly follow this idea and consider models as sequences of operations, because we can subdivide these with little overhead, as we show below. An UML-like model in operation-based formalization comprising an airport class looks something like this:

newPackage(‘*AirportModel*’) : newClass(‘*Airport*’) : newAttribute(‘*name*’)

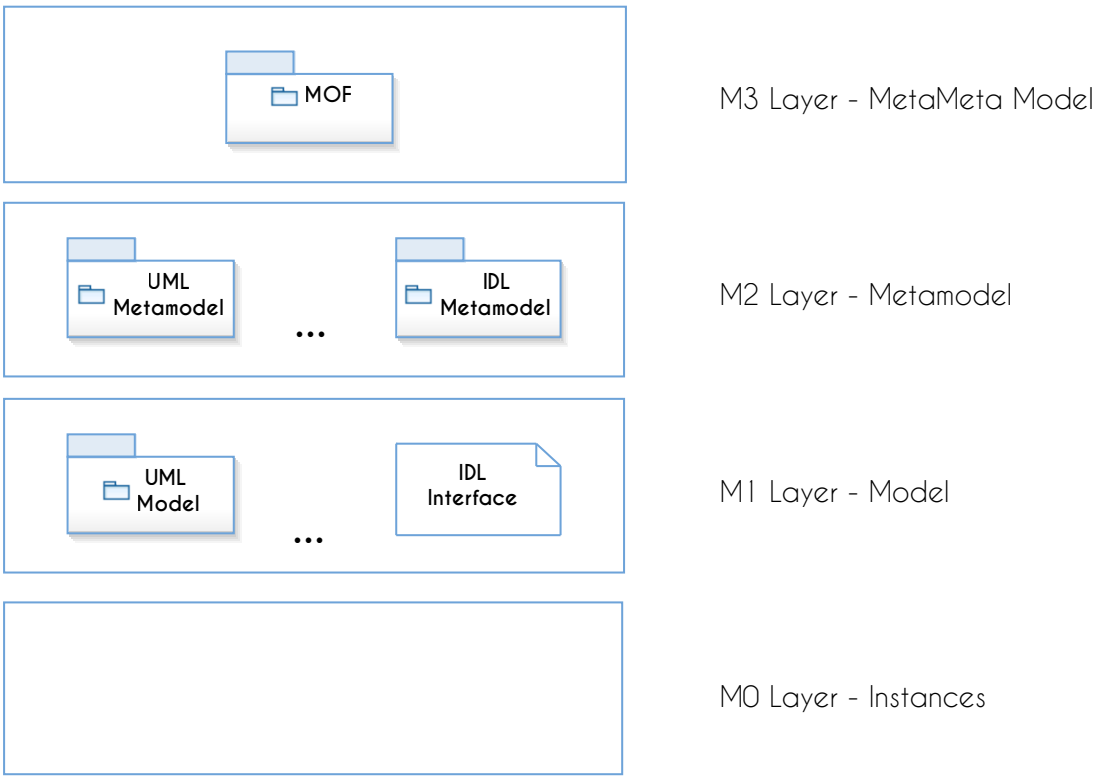


Figure 3.1.: Example for Metamodel Layers similar to [Obj02] dropped in [Obj14]

Note that this pseudocode notation only sketches the idea. Most importantly, we will later use the mathematical composition relation, which reverses the order of the operations to be read from right to left.

Subsequently, we gain operation-based models by, first, defining model elements as they appear in our target environment. Then, we use these model elements and provide semi-formal model operations as needed to support model evolution in section 3.4 and model reuse in section 3.5. While developing elements, relations, and operations, we postpone some design rationales and observations to subsection 3.1.4 for the sake of readability.

3.1.1. Model Elements and Relations

Operations in our framework will map elements from a domain to a co-domain, just as relations can do with real numbers. However, what could be our domain and co-domain for our operations on models and what could be our modeling layer (cf. figure 3.1)?

The modeling layer we intend to deal with is best depicted as M3 in figure 3.1. This is the topmost layer, named the meta-metamodel layer in the MOF 1.4.1 specification [Obj02]. It defines MOF and its basic elements of `Classifier`, `Instance` or `Class`, and `Object`, which are necessary to define [Obj14], e.g., the UML [Obj11a]. The detailed layering

shown in figure 3.1 was dropped in MOF 2.0, because the standard should not confuse users with a specific number. Now, the specification simply states that the number of abstraction layers is open-ended with a minimum of two [Obj14].

However, MOF as a language is considered to be complex and EMOF, the essential MOF, was introduced as a lightweight alternative. It is stripped to the minimum subset of MOF, which is still expressive enough for most scenarios. Additionally, the "Essential Meta Object Facility (EMOF) [...] quite closely resembles Ecore" [Ste+08], and some tools demonstrate how to transform back and forth between MOF and Ecore [GR03]. This is of vital importance for the course of this text, because Ecore, which is the metamodel of EMF, is the basis for our realization and is often seen as an implementation of EMOF. Hence, our solution works interchangeably in both with one limitation: a relation in Ecore metamodels denotes accessor methods [KE14], whereas in UML "[a]n association describes a set of tuples" [Obj14; Obj11a]. In other words, multiplicities are handled differently, but with congruent semantics. This difference is negligible for the definition of our domain and co-domain. Therefore, we can build them with help of the Ecore metamodel from the EMF handbook [Ste+08].

A reduced version of this metamodel, only depicting the relevant information for our definitions, is shown in figure 3.2. It comprises model elements, as coined by Wachsmuth [Wac07], and relationships. Accordingly, we semi-formally define sets for the depicted elements (E_x) as follows (cf. table 3.1): E_C is the set of all `EClasses`, E_A is the set of all `EAttributes`. Note that the element sets in italics are abstract in the sense of object orientation, which makes our definitions semi-formal. In our case, this means that they are automatically constructed unions from other sets, which are subelements in figure 3.2. For example, `EClass` and `EDataType` are subelements of `EClassifier`, and hence: $E_{cl} := E_C \cup E_D$ or `ENamedElements` is $E_{ne} := E_{te} \cup E_{cl} \cup E_{PK} \cup E_L$.

Table 3.1.: Element Sets in Ecore (cf. figure 3.2)

Element	Set-ID	Element	Set-ID	Element	Set-ID
Classes	E_C	Packages	E_{PK}	*Named Elements*[1]	E_{ne}
Attributes	E_A	Enums	E_E	*Classifiers*[1]	E_{cl}
Operations	E_O	Literals	E_L	*Typed Elements*[1]	E_{te}
Parameters	E_P	DataTypes	E_D	*Structural Features*[1]	E_{st}
References	E_R			*Model Elements*[1]	E_{me}

[1] Element sets in *italics* (and lowercase index) are abstract

In addition to the definitions derived from figure 3.2, we need the basic data types expected for $E_D := E_{Boolean} \cup E_{Integer} \cup E_{String} \cup \dots$, but note that these are mostly platform-dependent, e.g., in favor of Java. This breaks the interchangeability between EMOF and EMF/Ecore. Hence, we did not put them in table 3.1. In EMF/Ecore, these data types are introduced in models by E_D-elements. For example, an E_D-element called `JString`

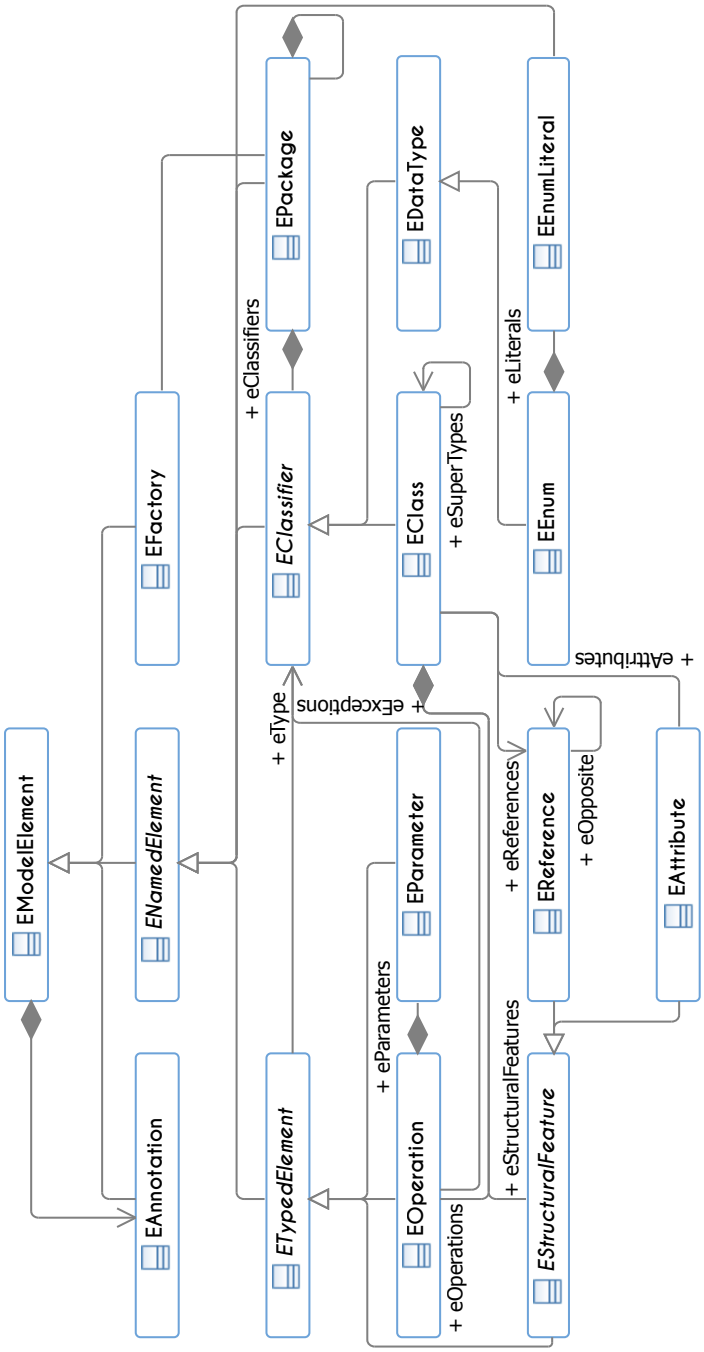


Figure 3.2.: EMF Ecore Metamodel Excerpt [Ste+08]

could refer to `java.lang.String` and then enable attributes of type `JString`, which are actually real Java strings.

Another "attribute", or compulsory property, which comes with each element in table 3.1 (cf. the complete figure 3.2 in [Ste+08]) is the name property. In Ecore, this belongs to the `ENamedElement` and is, therefore, part of all the elements we define in table 3.1. To be more precise, the elements become tuples of $(E_{ne} \times \text{String})$. For the sake of clarity and comprehensibility, we do not address elements as tuples, but use a superscript notation. As an example, an `EClass`, say $\varepsilon_C \in E_C$, with name property `Airport`, i.e., $(\varepsilon_C, \texttt{Airport})$, is denoted as $\varepsilon_C^{\texttt{Airport}}$. For contained elements, such as attributes, we use a path (breadcrumb) notation to maintain uniqueness. Consider, for example, an $\varepsilon_A \in E_A$ called `name`. This could be contained in every $\varepsilon_C \in E_C$. Therefore, we use the name property of the container element as a prefix, e.g., in our case `Airport`.*name*, or for the element $\varepsilon_A^{\texttt{Airport.}name}$. Note that the name property is required to identify or index, as we do in subsection 3.2.2.

Careful readers will have noticed that we omit the Universally Unique Identifiers (UUIDs) for all the elements. Of course, every E_{ne}-element has such an ID, but we try to keep our approach simple and readable. Regardless, bear in mind that this provides important information for identifying elements in models or renamed duplicates. For now, we consider it 'just' a realization detail.

Next to elements, figure 3.2 shows model relationships (ρ_x), only a few of which are named for reasons of clarity and comprehensibility. They are necessary to express how elements are related. For example, an `EClass` can be related to another `EClass` by means of an `eSuperType`. This is a notion for generalization between two classes, and might be expressed as an element in $\rho_{\text{eSuperTypes}}$ (cf. table 3.2). In the case of two `EClasses` named A and B, where A is the superclass of B, the relation is $(\text{B}, \text{A}) \in \rho_{\text{eSuperTypes}}$. Note that some relations have special properties, such as the relation $\rho_{\text{eParameters}}$. This is not binary, because an operation can take several parameters. We unite all relations for convenience, and denote them as: $\rho_e := \rho_{\text{eSuperTypes}} \cup \rho_{\text{eAttributes}} \cup \ldots$.

Table 3.2.: Relationship Sets in Ecore (cf. figure 3.2)

Relation-ID		Relation	Relation-ID		Relation
$\rho_{\text{eSuperTypes}}$	$\subseteq$	$E_C \times E_C$	$\rho^*_{\text{eExceptions}}$	$\subseteq$	$E_O \times E_{cl}$
$\rho^*_{\text{eAttributes}}$	$\subseteq$	$E_C \times E_A$	ρ^*_{eType}	$\subseteq$	$E_{te} \times E_{cl}$
$\rho_{\text{eOperations}}$	$\subseteq$	$E_C \times E_O$	$\rho_{\text{eStrucFeatures}}$	$\subseteq$	$E_C \times E_{st}$
$\rho_{\text{eParameters}}$	$\subseteq$	$E_O \times (\times E_P)$	$\rho_{\text{eClassifiers}}$	$\subseteq$	$E_{PK} \times E_{cl}$
$\rho^*_{\text{eReferences}}$	$\subseteq$	$E_C \times E_R$	$\rho_{\text{eLiterals}}$	$\subseteq$	$E_E \times E_L$
$\rho^*_{\text{eOpposite}}$	$\subseteq$	$E_R \times E_R$			

Considering tables 3.1 and 3.2, a model of all models ($m \in \mathcal{M}$) is a tuple of sets:

$$m := (E_A, E_{cl}, E_C, \ldots, \rho_{\text{eAttributes}}, \rho_{\text{eClassifiers}}, \ldots, \mathfrak{R}), \quad \mathfrak{R} : \text{constraint rules} \tag{3.1}$$

As an example for $\mathfrak{R}$, figure 3.2 states that an `EEnum` can only contain `EEnumLiterals`. These constraint rules are only mentioned for completeness, they do not play any further role. Put together, an excerpt of the model from figure 2.3 (p. 27) without a container package but comprising the classes `Airport` and `Runway` could look like this (see elements and relationship sets in table 3.3):

$$\mathsf{m}_{\mathit{Ecore}}^{\mathit{figure\ 2.3}} := (\mathsf{E}_{\mathsf{A}}^{\mathit{figure\ 2.3}}, ... \mathsf{E}_{\mathsf{C}}^{\mathit{figure\ 2.3}}, ... \mathsf{E}_{\mathsf{R}}^{\mathit{figure\ 2.3}}, ... \rho_{\mathsf{eAttributes}}^{\mathit{figure\ 2.3}}, ... \rho_{\mathsf{eReferences}}^{\mathit{figure\ 2.3}}, ...) \quad (3.2)$$

Table 3.3.: Sets build on excerpt from figure 2.3 (p. 27)

Set-ID		Element Set	Relation-ID		Relation Set
$\mathsf{E}_{\mathsf{C}}^{\mathit{figure\ 2.3}}$	=	{`Airport`, `Runway`}	$\rho_{\mathsf{eAttributes}}^{\mathit{figure\ 2.3}}$	=	{(`Airport`, `Airport.`*name*), (`Airport.`*name*, $\mathsf{E}_{\mathsf{String}}$)}
$\mathsf{E}_{\mathsf{A}}^{\mathit{figure\ 2.3}}$	=	{`Airport.`*name*}	$\rho_{\mathsf{eReferences}}^{\mathit{figure\ 2.3}}$	=	{(`Airport`, `Airport`*2*`Runway`)}
$\mathsf{E}_{\mathsf{R}}^{\mathit{figure\ 2.3}}$	=	{`Airport`*2*`Runway`}	$\rho_{\mathsf{eType}}^{\mathit{figure\ 2.3}}$	=	{(`Airport`*2*`Runway`, `Runway`)}

The sets introduced above require further remarks in some respects, because we omitted some details from figure 3.2 for the sake of simplicity. First, we ignore the relationship from `EPackage` to `EPackage`, because we work solely on a flat-single-package level and do not need subpackages. Second, we skipped `EAnnotations` entirely, as they do not provide any semantic benefit for our approach. Finally, our topmost element type is `ENamedElement` and not `EObject`. `EObject` in EMF/Ecore is like `Object` in Java, the root object of all objects. We could map `ENamedElement` to `EObject`, but that would be an erroneous interpretation of the EMF runtime environment that maps `EModelElements` to `EObjects`. Altogether, these simplifications can be adjusted quickly if needed (cf. [KE14]).

3.1.2. Model Operations

The elements and relations introduced above are suitable for status descriptions of models. We have demonstrated that with an example comprising an excerpt of figure 2.3 (p. 27). However, this format does not provide any information about the order of editing steps used to create the model, unless we compare two other consecutive status descriptions. An approach preserving each editing step is possible when changing the perspective to an operation-based view. We do so subsequently, using the elements and relations introduced above as the foundation for deriving our approach from standards.

The operation-based view on models, which we use throughout this text, uses the MOF reflective API as a starting point [Obj14]. This provides an idea of a reasonable set of operations, which can be carried out on models. These work similarly on Ecore models, because the MOF reflective API was also designed for EMOF and we use Ecore as an implementation of EMOF, as explained above. This means that we take the operations defined for MOF and create a conceptual layer for our approach, which works seamlessly on top of the implemented Ecore API of EMF in the Eclipse platform [Vog13; Ste+08].

In detail, the combined Complete MOF (CMOF)/MOF reflective API defines operations to create, get, set, and unset elements. The create operations are provided by a factory (cf. design patterns [Gam+95]) for elements, which are similar to our elements, and links akin to our relationships. Regarding additional properties, both are treated as objects, so they provide the get, set, and unset operations. This deviates from CRUD [Mar83], but we keep to this notion and define model operations semi-formally:

$$\Pi := \{\Pi_b \cup \Pi_s\}, \quad \Pi_b : \text{Basic Operations}, \Pi_s : \text{Supporting Operations} \tag{3.3}$$

This means that Π in equation (3.3) is our set of semi-formal model operations, which fall into two subsets. First, the set Π_b comprises basic operations with elementary operations to alter a model. Second, the set Π_s contains supporting operations, which are needed to simplify or rename basic operations. In more detail, Π_b is constructed in equation (3.4) using a create π_c, set $\pi_{p^{set}}$, associate $\pi_{r^{ass}}$, and delete π_d operation. Further, Π_s is defined in equation (3.9) with a typing Γ, find φ, update π_u, and revert π_{-1} operation.

$$\Pi_b := \{\pi_c, \pi_{p^{set}}, \pi_{r^{ass}}, \pi_d\} \qquad \textbf{Basic Operations} \tag{3.4}$$

$$\pi_c := \mathcal{M} \times \mathsf{E}_{ne} \to \mathcal{M} : \qquad (\mathsf{m}, \varepsilon^{\text{NAME}}) \mapsto \mathsf{m}',\ \mathsf{E}_{\Gamma(\varepsilon)} \cup \{(\varepsilon, \text{NAME})\} \tag{3.5}$$

$$\pi_{p^{set}} := \mathcal{M} \times \mathsf{E}_{te} \times \bigtimes \mathsf{E}_{ne} \to \mathcal{M} : \qquad (\mathsf{m}, \varepsilon, \nu) \mapsto \mathsf{m}',\ \nu \text{ properties} \tag{3.6}$$

$$\pi_{r^{ass}} := \mathcal{M} \times \mathsf{E}_{\mathsf{R}} \times \bigtimes \mathsf{E}_{ne} \to \mathcal{M} : \qquad (\mathsf{m}, \varepsilon, \nu) \mapsto \mathsf{m}',\ \nu \text{ destinations} \tag{3.7}$$

$$\pi_d := \mathcal{M} \times \mathsf{E}_{ne} \to \mathcal{M} : \qquad (\mathsf{m}, \varepsilon^{\text{NAME}}) \mapsto \mathsf{m}',\ \mathsf{E}_{\Gamma(\varepsilon)} \backslash \{(\varepsilon, \text{NAME})\} \tag{3.8}$$

Regarding the individual model operations, several things should be noted. Our π_c from equation (3.5) works as a general producer (or factory) for any type of E_{ne}-elements and takes a parameter in superscript notation, as introduced above. Contrasting with the MOF reflective API, one operation is responsible for creation instead of two. Moreover, this operation is semi-formal because it includes a creation process, often referred to as instantiation [Bor07; Her11; Rom+07], which we do not define in more detail.

$$\Pi_s := \{\Gamma, \varphi, \pi_u, \pi_{-1}\} \qquad \textbf{Supporting Operations} \tag{3.9}$$

$$\Gamma := \mathsf{E}_{ne} \to \text{TYPE} : \qquad \varepsilon \mapsto \{\text{EAttribute}, \ \dots\} \tag{3.10}$$

$$\varphi := \mathcal{M} \times \text{TYPE} \times \text{STR} \to \mathsf{E}_{ne} : \qquad (\mathsf{m}, \gamma, \text{ID}) \mapsto \varepsilon^x_\gamma,\ x = \text{ID}, \varepsilon^x \in \mathsf{E}_{ne} \tag{3.11}$$

$$\pi_u := \mathcal{M} \times \mathsf{E}_{ne} \times \bigtimes \mathsf{E}_{ne} \to \mathcal{M} : \qquad (\mathsf{m}, \varepsilon, \nu) \mapsto \mathsf{m}',\ \text{aka: } \pi_{p^{set}}, \pi_{r^{ass}} \tag{3.12}$$

$$\pi_{-1} := \mathcal{M} \times \Pi \to \mathcal{M} : \qquad (\mathsf{m}, \pi) \mapsto \mathsf{m}',\ \text{revert} \tag{3.13}$$

We do not delve into the details of instantiation, but introduce a semi-formal supporting operation instead. This operation Γ from equation (3.10) returns the type information of a provided parameter. Using this, π_c adds a new element to the correct set of E_γ-elements; i.e., the type of the new element (γ) determines the set to which it is added.

Another supporting operation is required for operations that expect existing elements as inputs. For example, $\pi_{r^{ass}}$ in equation (3.7), which acts to assign the referred ends

of an association, is such an operation. Hence, we need a supporting operation that takes a query string and returns an E_{ne}-element. Consider as an example $E_C^{\textit{figure 2.3}}$, as defined in table 3.3. We refer to an element in superscript notation, such as $\varepsilon_C^{\texttt{Airport}}$. Then, $\varphi(m, E_C, \text{Airport})$ should map to this exact class. Note that this is possible because $\varepsilon_C^{\texttt{Airport}}$ is short for $(\varepsilon_C, \text{ Airport})$ and the second part of the tuple uniquely identifies the tuple because of how we have set the naming conventions. For example, each *name* attribute in a class, as in table 3.3 for `Airport`.*name*, is referred to with the path notation introduced above. Thus, we can introduce a find operation as φ to serve our needs in equation (3.11). Again, we introduce a superscript notation as follows:

$$\varphi_\gamma^{\text{QUERYTERM}} := \varphi(m, \gamma, \text{QUERYTERM}) \tag{3.14}$$

In addition, we introduce two more supporting operations for convenience. First, an update operation (π_u) redirects to $\pi_{p^{set}}$ and $\pi_{r^{ass}}$. All that happens in between is to determine which operation to direct. This is supported by our type determining operation Γ. Second, a revert operation (π_{-1}) undoes the last operation. We do not define this function in more detail here, but mention it to provide a complete picture. The semantics of this operation are as expected.

3.1.3. Model-Operation Sequences

We can concatenate sequences of operations to rather long sequences, so it makes sense to introduce some convenient concepts. Operation sequences (σ) summarize model operations that are carried out successively and which are read from right to left. Note that, contrary to mathematical convention (cf. section 2.2 (p. 14)), they allow multiple parameters.

$$\sigma := \bigcirc_{i=0}^{n\in\mathbb{N}} \pi_i(m) = \pi_n \circ \ldots \circ \pi_0(m) \qquad \pi_i \in \Pi, \text{ and } m = \{\varnothing, \ldots\} \in \mathcal{M} \tag{3.15}$$

Further, we can compose sequences into sequences of sequences, comprising an arbitrary but finite number of sequences: $\bigcirc_{i=0}^{n\in\mathbb{N}} \sigma_i = \sigma_n \circ \ldots \circ \sigma_0$. For example, we consider operation sequences that are beneficial for the semantic grouping of operations. Thus, one sequence could contain operations for creating a package, and two more sequences could contain the operations that create a class each and their attributes. We will make use of this to semantically group model operations, but essentially consider it as syntactic sugar. We introduce a final convenient notation called operation-based models, denoted by a rotated capital sigma, and a respective model universe ($\beth \in \mathcal{M}^{\beth}$):

$$\beth := \bigcirc_{i=0}^{n\in\mathbb{N}} \sigma_i(m) \tag{3.16}$$

Example: Putting together our sets and operations, we can present an example with a stepwise operation-based view for the example presented in table 3.3. We skip the

operations for creating a basic package and omit the first parameters in our operations, because they are obvious. Thus, let $\mathrm{m} := (\mathrm{E_A}, \mathrm{E}_{cl}, \mathrm{E_C}, \ldots, \rho_{\mathrm{eAttributes}}, \rho_{\mathrm{eClassifiers}}, \ldots, \mathfrak{R})$ be an empty model. We first create the classes and attributes providing the name property in superscript notation (remember that "$\circ$" reads from right to left):

$$\sigma_1 := \pi_c(\mathrm{E_A^{\texttt{Airport.}\textit{name}}}) \circ \pi_c(\mathrm{E_C^{\texttt{Runway}}}) \circ \pi_c(\mathrm{E_C^{\texttt{Airport}}})$$

This created two classes with name properties `Airport` and `Runway`, next to an attribute `Airport.`*name* in path notation, as introduced above. Note the difference between the name property for the class and the attribute. The latter is meant to contain the actual name of the airport, whereas the former provides the type information. Next, we create a relationship element for our reference:

$$\sigma_2 := \pi_c(\mathrm{E_R^{\texttt{Airport2Runway}}})$$

Now, we have a relationship to put our elements together, and we can wire the `Airport` and `Runway` class. Observe that we now stop modeling the `Airport.`*name* with $\mathrm{E_{String}}$. From this point on, the runtime takes responsibility, because the name is not set at design time. Regardless, we first need to put our parts together:

$$\sigma_3 := \pi_{p^{set}}(\varphi_{\mathrm{E_C}}^{\texttt{A...}}, \{\varphi_{\mathrm{E_R}}^{\texttt{A...2Runway}}\}) \circ \pi_{p^{set}}(\varphi_{\mathrm{E_C}}^{\texttt{A...}}, \{\varphi_{\mathrm{E_A}}^{\texttt{A....}\textit{name}}\}) \circ \pi_{p^{set}}(\varphi_{\mathrm{E_A}}^{\texttt{A....}\textit{name}}, \{\varphi_{\mathrm{E_D}}^{\mathrm{E_{String}}}\})$$

At this point, we have an `Airport` class containing an `Airport.`*name* attribute of type `String` and an `Airport`*2*`Runway` reference with no target. This means we need to associate it with the actual destination class `Runway`:

$$\sigma_4 := \pi_{r^{ass}}(\varphi_{\mathrm{E_R}}^{\texttt{Airport2Runway}}, \{\varphi_{\mathrm{E_C}}^{\texttt{Runway}}\})$$

We now have a representation of table 3.3 and figure 3.3 as an operation-based model ($\mathfrak{Z}$). Subdividing the necessary steps into four operation sequences, we can put together a sequence of sequences (cf. equation (3.17)). Further, to improve readability, we can rename the sequences to provide a summary of what each of them does (equation (3.18)):

$$\mathfrak{Z}_{\textit{Ecore}}^{\textit{figure 2.3}} = \bigcirc_{i=1}^{4} \sigma_i(\mathrm{m}) = \sigma_4 \circ \sigma_3 \circ \sigma_2 \circ \sigma_1(\mathrm{m}) \tag{3.17}$$

$$= \sigma_{\textit{relationships}} \circ \sigma_{\textit{properties}} \circ \sigma_{\textit{references}} \circ \sigma_{\textit{basic elements}}(\mathrm{m}) \tag{3.18}$$

3.1.4. Design Rationales and Observations

The model elements, relations, and operations introduced above address the "Representation Challenge" mentioned in section 1.2 (p. 6) and follow the design rationales we have sometimes omitted for the sake of simplicity. For a coherent picture, though, we consider these rationales vital, and provide them subsequently as follow-ups. Further, we present some more observations in necessarily dry form.

The overall goal in developing operation-based models is to obtain a notation that is

both comprehensible and sufficiently formal to address two application cases. First, the notation should foster descriptions of editing sequences built from persisted models, so they can be applied within editors step by step, as a human would do. This could mean transforming models as XMI-documents, or more precisely, their document object models, into operations (denoted m $\rightsquigarrow \Im$). Second, the notation should be able to represent the editing steps carried out by a human as they are captured during modeling. We imagine this as being similar to a macro recorder for GUIs. At the same time, the notation should hide obvious and simple details. Hence, we came up with a semi-formal notation fulfilling these requirements.

Careful readers will already have noticed that some EMOF/Ecore aspects are not expressible with our operations. The details we have omitted are instantiation, cardinalities, containments, and abstraction. We did so for simplicity reasons, and we are certain that extensions supporting these details could be developed quickly. For example, the latter three are properties of relationships, which could be quickly set by additional supporting operations. For instantiation, we admit the extension will be more complicated and refer to the work of Boronat and Meseguer, Koshima and Englebert, and Boronat for further details [BM08a; KE14; Bor07]. In our cases, instantiation does not add any benefit.

Furthermore, we can, at best, call our notation semi-formal, because we neglected to discuss types and the details of our abstract and concrete sets. To keep things simple, we work on concrete sets like E_C or E_A and say that the supersets such as E_{cl} and E_{st} are automatic unions of the subsets. Detailed discussions and definitions on type hierarchies for modeling or the metalanguage ML are provided by, e.g., Alanen and Porres, Boronat, Leroy, Herrmannsdörfer, or Milner [AP07; Bor07; Ler06; Her11; Mil78]. Further, we changed the topmost element to `ENamedElement` and did not discuss UUIDs at all. Moreover, we only introduced π_d as a deleting operation without going into any detail. The issue with this is that certain element deletions could trigger cascades of deletions, which we do not address on a conceptual level. However, deletions only play a minor role in capturing modeling activities and can be mostly neglected in our concept. The same holds true for the undo operation, π_{-1}, introduced above. Finally, related to cascading deletions, our set of constraint rules ($\mathfrak{R}$) is only mentioned for completeness. We can do so because the underlying EMF editing framework keeps the model in a reasonable state at runtime, as we will discuss later. This means that the potentially "broken references" caused by deletions are not an issue in our case. Finally, our find operation, φ, does not cover the case in which the queried element is not found or not available. This could lead to erroneous $\pi_{p^{set}}$ operations and untyped attributes or erroneous $\pi_{r^{ass}}$ operations, which could result in dangling references.

Next to operations, we sketched several notations for models in the introduction to this chapter, but opted for operations. The reason for this, next to sequentiality, lies in the experience gained with operations since they were proposed by Lippe and van Oosterom for merging textual documents [Lv92]. This keeps the door open for conflict handling in merging scenarios, because many approaches have since been developed, e.g., in the area of model evolution by Kögel [Kög11]. Moreover, we could derive our operations from the EMOF reflective API, which is a well-established foundation for model manipulation,

as demonstrated by several implementations such as EMF/Ecore. In addition, similar attempts using the same paths exist for manifold scenarios, e.g., by Berlage and Genau for document interaction histories [BG93], or by Blanc et al. for inconsistency checks and Mougenot, Blanc, and Gervais for collaborative editing support [Bla+07; Bla+09; MBG09]. Further, logic-based operations, to our mind, do not seamlessly integrate in our EMOF, Ecore environment, and seem to suffer from non-terminating issues regarding UML, as demonstrated by Mougenot, Blanc, and Gervais [MBG09]. Similarly, van der Straeten, Ragnhild et al. require a conceptual break for their logistic formula-based approach [van+03].

Regarding the structure of models, we have not discussed two properties of our models. First, our elements and relationships describe a linked tree, or rather graph structure, which can be derived by regarding the elements from table 3.1 as vertices and the relationships from table 3.2 as edges. The root element is a package containing classes and enumerations. Note that this is mostly assured by our rule set. Second, relationships are possible, by definition, only within one model. This means that, in UML terms, only internal references are allowed. In other words, we can associate classes as usual within one model, but ban associations with other files, i.e., prohibit distributed models, sometimes called “clouded models”. We designed this on purpose to ensure only uncoupled models, and introduce an extension to this later in section 3.2. The usual way would be to introduce Uniform Resource Identifier (URI) for element prefixes. Note that there are different kinds, e.g., a Java URI differs from a EMF URI [Ste+08].

Finally, note that we work on flat and uncoupled models only and use a path/breadcrumb naming convention! The former means that we do not need subpackages or external references and can put all elements of a model in one package. The reason we can set this constraint lies in the intended purpose of our models for reuse in an operation-based manner only. Hence, models need to remain relatively small so they can be reviewed in a short period of time (roughly a few seconds). Larger models of, say, several dozen elements need too much time for review. The path/breadcrumb notation means that we use a naming convention for elements to simplify the readability and establish uniqueness. For example, every class could hold a name property that would not be unique in our case. As we avoid using UUIDs, we prefix contained elements with the surrounding classifier name, which is often the name of the container class followed by a dot. Note that this cascades.

3.1.5. Related Work

The foundations of our presented approach comprise model elements and operations. Both have undergone massive term overload, and some early work could be considered related. For example, Tai explains the editing distance, i.e., similarity, between two labeled ordered trees with operations, e.g., *delete* [Tai79], similar to our approach. Still, we consider this approach unrelated. Hence, we limit the scope of related approaches and present only exemplary approaches as representatives. In doing so, we subdivide model elements and operations and discuss which ideas influenced our work.

Modeling Elements

Amelunxen and Schürr provide a formal definition for MOF 2.0 [AS08]. Their metamodel definition, MM:=(C, A, *firstEnd*, *secondEnd*), comprises two sets and binary associations. C is a finite set of class identifiers and A is a finite set of association identifiers. Further, *firstEnd* is a relation that returns the first class of a given association identifier, *secondEnd* the second, and so on. This allows the bootstrapping of metamodels and models, i.e., M:=(O, L, *class*, *association*, *firstObject*, *secondObject*). Here, M represents a model that is roughly related to a given metamodel as follows (for details, see [AS08]): O are Objects represented as object IDs and L associates objects, just as A does with classes. Consequently, *firstObject* and *lastObject* work similarly to the binary associations above. Finally, *class* relates object IDs to classes, i.e., elements $\in C$. Altogether, this notation is much smaller and more efficient than our notation, but a closer look unveils that both are congruent if we consider all concepts and all relationships as one set each. We have simply opted for a more readable notation, so we can more easily distinguish between concepts and provide an easily comprehensible form. However, the similarities continue in regard of operations and they formalize MOF 2.0 entirely, even considering multiplicity, inheritance, and so forth. In our case, these details provide no benefit and so we did not include them. Instead, we added supporting operations as needed and introduced operation sequences. Another formal notation provided for UML 2.0 in the form of a system model for structural data is provided by Broy, Cengarle, and Rumpe [BCR06]. Kim and David and Shroff and France provide an alternative for UML class structures formulated in (Object) Z [KD99; SF97].

One approach that investigates model merging is Westfechtel [Wes14]. This builds on a state-based idea instead of an operation-based idea. Still, Westfechtel must define a model formally, and do so in a more simplified manner than we do, focusing on the properties of relationships, e.g., cardinalities. Moreover, as this approach is focused on model merging, the author is more concerned about object IDs than we are.

Boronat and Meseguer employ membership equational logic as the basis for their formal algebraic semantics of MOF 2.0 [Bor07; BM08a]. This makes their theory executable, which is beneficial for formal analysis. Eventually, they provide an implementation in Maude [cf. Rom+07, Maude]. Altogether, membership equational logic requires more heavy lifting than our approach, and we do not go into more detail about this.

Operation-Based Approaches

Looking at data in an operation-based manner is not a novelty, and we can only discuss a very limited set of related approaches. We distinguish them by the data on which they operate. Overall, we look into source code, databases, grammar, and models, although they cannot always be distinguished as simply as this; source code, database schema, and grammar can be considered models, after all.

Source Code: Combining two or more software artifacts, such as source code or textual documents, is often studied in terms of the operation-based merging of these artifacts.

Lippe and van Oosterom claim to have introduced operation-based merging for "parallel development activities" in object management systems (OMS), and this "can in principle be applied to arbitrary abstract data-types, and guarantees that data-type invariants are respected" [Lv92]. The developed system, called CAMERA, detects conflicts in merging scenarios. It does so by recording operations as (primitive) transformations and forwarding them to an OMS, i.e., object-oriented database. This fosters parallel development, because the recorded transformations can be applied to a remote system. Though Lippe and van Oosterom do not discuss the details of the involved operations and we omit the issues associated with merging, their idea of recording and forwarding operations is very similar to what we eventually do; however, our purpose and the approaches between recording and forwarding are completely different, as we show in section 3.3, 3.4, and 3.5.

About a decade after Lippe and van Oosterom introduced operation-based merging, Mens discussed the state of software merging, stating that "there is a tendency towards operation-based [software] merging because of its increased expressiveness" [Men02]. This is because, compared to state-based approaches, syntax and semantics are taken into account. Moreover, the author regards operation-based approaches as flavors of change-based approaches.

An example for a change-based approach is presented by Berlage and Genau [BG93]. It works by recording operations as transformations, which can be put together to form sequences. They enable multiple undo/redo mechanisms because they capture the "interaction history of a document", here called the command history, as a tree of command objects. This is similar to our approach, but we work in a different scenario. While they work in versioning with branching and merging, we focus on model reuse by recommender systems. Still, the command history, neglecting Prolog, is almost the same as our editing sequences for our recommender (cf. subsection 3.5.4 (p. 123)).

Steyaert et al. discuss reuse in the context of operation-based merging with reuse contracts [Ste+96]. Their approach concerns compiled artifacts and when to recompile them. For example, clients or derived classes might not need recompilation, even if a base class interface is modified and recompiled. This is achieved through "reuse contracts and their operators [which] facilitate the propagation of changes". In detail, the mentioned operations are extension, refinement, and concretization. This expresses rather high-level operations on preexisting artifacts, but shows how reuse and an operation-based approach can work together.

Databases: An early example for an operation-based approach for relational databases was proposed by Shneiderman and Thomas [ST82a; ST82b]. They propose a set of fifteen transformations to be applied to schema and their respective data. Another early example designed for hierarchically ordered tabular data was proposed by Shu, Housel, and Lum [SHL75] and Shu et al. [Shu+77]. They propose a data definition language called DEFINE and a transformation language called CONVERT, which they demonstrate in a prototype named EXPRESS. The realization relies on PL/1 and "compiles a set of PL/1 procedures" from the CONVERT descriptions.

Relating to databases, Ambler and Sadalage integrate recent developments in com-

puter science into operation-based migrations on databases [AS06]. They subdivide several rather high-level refactorings, which we can consider as operations, according to the following categories: *architecture*, e.g., introduce index, *structural*, e.g., merge tables, *data quality*, e.g., drop non-nullable data, *referential integrity*, e.g., add foreign key constraint, *method refactorings*, e.g., remove method, and *transformations*, e.g., insert or update data.

A workbench dedicated to database evolution was proposed by Curino et al. [Cur+09]. A key concept in this tool is a set of "Schema Modification Operations (SMO)", which include the create and drop table operations. The set of SMOs enables "Logical Mapping" and "Query Rewriting" [CMZ08]. The first of these is derived from SMOs capturing constraints and relationships between successive versions of database schema in a notation called "Disjunctive Embedded Dependencies (DED)". Query Rewriting uses the information in DED notation to rewrite and update queries.

Casais introduces a taxonomy developed for transformations in object-oriented databases (OODB) [Cas95]. These transformations, e.g., *insertion of an attribute*, are derived from four exemplary OODB and discussed with respect to their impact on the database schema in ORION [cf. Ban+87], GemStone [cf. PS87], and O_2 [cf. Fer+95]. Further, restructurings and refactorings are discussed on these grounds. In a sense, this is very similar to model evolution and co-evolution, as we see below, if we take the actual database with instances as the model and the schema as the metamodel with classes, inheritance relationships, and other properties. In our case, the set of operations is of interest because it is congruent to ours, but is only provided in textual form.

Similarly to Casais and the methods he employed, Brèche introduces advanced primitives for OODB schema [Brè96]. These include *merge*, *aggregate*, and *remove*, which are all defined by low-level primitives (LLPs) such as *add a new method*, defined by Zicari for OODB schema [Zic91]. The LLPs provide a validation set for our operations, because they are provided textually. Further, the advanced primitives show application scenarios that we can proofread our operations against.

Models: With models and metamodels in mind, several areas of operation-based approaches have been studied. Issues of model evolution, co-evolution, collaborative editing, or inconsistency detection have been examined, with aspects of algebraic foundations introduced as foundations.

Rose et al. analyze the state of operator-based and other approaches for model evolution, which is essentially operation-based from our point of view [Ros+09]. Their point is that operator-based approaches define a set of operators that form a library for model evolution and co-evolution. Further, they discuss the usefulness of such libraries regarding their richness versus their complexity by referring to a delete example in work by Lerner [Ler00]. Our point is that we organize operations in sets and subdivide them regarding their purpose. Further, Lerner's compound operations are similar to our sequences, but Lerner operates on database schema. With respect to richness and complexity, we do not face the impact analysis issues that they do for co-evolution.

Wachsmuth looks at research conducted in the area of grammar engineering and ports it to model adaptation [Wac07]. He works on Petri nets and demonstrates how adoptions

in models and metamodels require co-adoptions. The foundation for his work is MOF 2.0/QVT, and he uses the term concepts for the elements in metamodels. Altogether, he proposes operations grouped in sets of refactoring, construction, and destruction, which are richer than our operations, but he implicitly uses a similar set of basic operations, like *introduce class* or *eliminate property*.

Kögel researched model evolution issues in collaborative editing settings [Kög11]. Hence, the tool implementing his approach tracks model changes and forwards them to a model evolution control system. The tracks are held in an operation-based manner, so conflict detection and resolution are enabled in merging scenarios. The foundation for this is an operation-based transformation algebra comprising seven operators, e.g., *application operator* or *conflict operator* and five axioms, such as *element creation* or *cancellation*. Overall, his approach addresses issues of merging that differ from our requirements. Hence, we do not need a conflict operator, cancellation axiom, or even a Boolean algebra as a foundation. In our case, it is more important that we can record and alter operations.

Herrmannsdörfer investigates model evolution and co-evolution [Her11]. He distinguishes between primitives and operations. The former have a structural or non-structural nature, e.g., *create package* and *rename*, respectively. Further, he needs coupled operations for co-evolution aspects and introduces sequential composition. This is all conceptually similar to our approach, although tailored for model evolution. Further, he continues and classifies sequences into reusable coupled operations. Altogether, his approach makes up a library of more than sixty operations grouped in seven sets.

Blanc et al. use operation-based model construction to detect model inconsistencies [Bla+07; Bla+09]. Therefore, they derive operations from the CMOF reflective API, just as we do, but they feed derived rules and facts to Prolog, unlike us. Their check engine on top of Prolog can detect structural or methodological inconsistencies. The latter might be a feature that was created before a corresponding requirement was created. Similar to that, van der Straeten, Ragnhild et al. use logic formulae to maintain consistency between models [van+03]. They built an EvolutionTrace and an HorizontalTrace to prevent “incompatible behavior conflicts” using description logics.

Grammars: Designers of programming languages like Java or C++ try to avoid grammar changes between releases. In the area of domain specific languages, this is different: migration paths are needed, and some of them are operation-based.

Pizka and Jurgens proposed a divide and conquer approach in their language evolution concept [PJ07]. This allows multistep rather than one-step compilation and fosters “improved structuring of DSLs”. Further, their “Grammar Evolution Language (GEL) comprises statements to declare nonterminals, to create, rename, and delete productions, to add, modify, and remove (literal, terminal, or nonterminal) production components and (inherited or synthesized) attribute declarations, to set semantic rules, to change the order of production components, and to influence priorities and the associativity of productions.” This is far more complex than our operation set, because not only do elements need to be considered, but there are also semantic rules and nonterminals. In contrast, we need to consider relationships in more detail.

Su et al. introduce a taxonomy for operation-based Extensible Markup Language (XML) evolution [Su+01]. They operate on graph structures and use primitives like *createDTDEl* or *destroyDTDEl* with formally defined preconditions. Further, they derive operations like *add-attribute* from their primitives. They claim to provide a sound but minimal set of primitives for Document Type Definitions (DTDs) and XML documents, which preserves the well-formedness and validity of DTDs and their related XML documents. This puts their work in between databases, models, and grammars, because DTDs, currently XMLSchema, could be considered as a grammar or schema and DTD-based XML documents could be seen as databases or models, e.g., XMI documents for EMF/Ecore. Essentially, their set of six operations is not expressive enough for our purposes, because no relationships are considered.

3.1.6. Summary of Operation-Based Models

This section has laid out the foundation for how we will develop operation-based model recommenders. These foundations comprise a semi-formal notation for the model elements and model relationships we need for modeling. Among them is a set of `Classes` denoted as E_C or a set of `Attributes` referred to as E_A. For relationships, we introduced relations like $\rho_{\text{eSuperTypes}}$ for relating classes, one for the supertype and one for the subtype. Altogether, this enabled a notation for models as tuples of sets $m := (E_A, E_{cl}, E_C, \ldots, \rho_{\text{eAttributes}}, \rho_{\text{eClassifiers}}, \ldots, \mathfrak{R})$ of a model universe ($\mathcal{M}$) containing a set of rules ($\mathfrak{R}$) that we only mentioned for the sake of completeness. Further, these foundations comprise basic operations, which we defined on top of these sets to create elements (π_c) or modify them ($\pi_{p^{set}}$). Moreover, supporting operations help to find (φ) elements or determine types of elements (γ). For convenience, we introduced a super and subscript notation as shortcuts. For example, $\varphi_{\gamma}^{\text{QUERYTERM}}$ looks up a name as given in the QUERYTERM and delimits it by the type provided as γ. Similarly, the found element is denoted as $\varepsilon_{\gamma}^{\text{NAME}}$. In addition, we introduced sequences of operations, which provide syntactic sugar and enable operation-based models denoted as $\mathfrak{I}$. Finally, we demonstrated the approach with an example (cf. figure 3.3), and discussed design rationales as well as shortcomings.

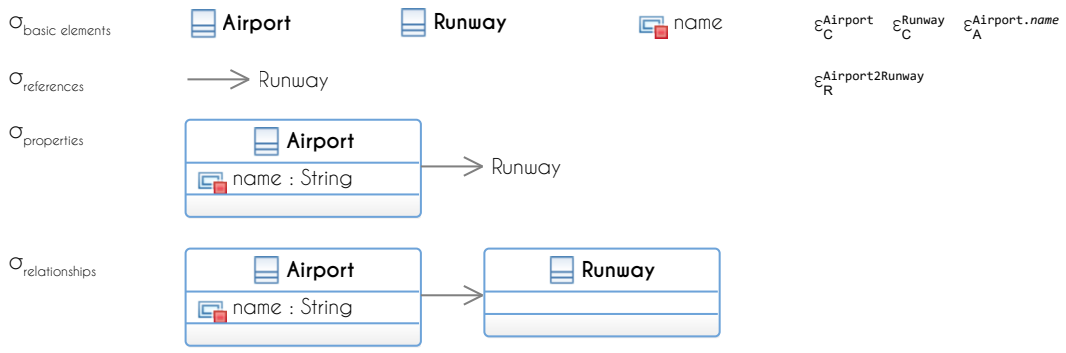
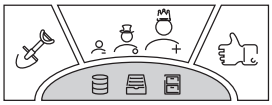


Figure 3.3.: Sketch illustrating Operation Sequence from equation (3.18) (p. 40)

3.2. Storing Models



One cornerstone for model reuse is a suitable organization for storing them, as introduced in section 1.2 (p. 6) as the representation challenge. This goes beyond questions of persistence and distribution. Related considerations comprise questions of the modeling relationships between models, of extracting information from models for indexing and retrieving them, and of versioning.

Some of these aspects were researched as model repositories or global model management. The former mostly concerns versioning or collaborative editing scenarios [Kög11], whereas the latter often focuses on project-wide model relationships and change impact analysis [All+06; Béz+05]. Consequently, models are treated as project assets, but they are not seen in the bigger picture, i.e., their greater potential for reuse is often neglected.

In other fields of software development, general solutions are common practice. For example, object-oriented design patterns became popular in object-oriented design [Gam+95]. Today, they are used as a set of principles and as a common language among developers [ACS13]. Similarly, best practices are often sought as reusable assets, and, if found, considered valuable [LR07]. For example, many architectural styles are considered best practices among architects [TDM10], and refactorings are seen as methodological best practices among programmers [Fow99].

In the case of domain knowledge, efforts toward identifying and storing reusable assets have faded. Some approaches apply ideas from machine learning techniques [BMM09; WKB09], ontologies [Obe14b; UG96], or artificial intelligence [MTM09]. However, how to organize, represent, and retrieve domain knowledge remain open questions [JHA14].

We agree with Chaudron, Heijstek, and Nugroho and consider models as an effective means of illustrating domain knowledge [CHN12]. Additionally, we think that UML class diagrams provide a good foundation for domain modeling. Therefore, we seek answers to the questions mentioned above for domain knowledge modeled as class diagrams. An example for such a model is depicted in our running example in figure 2.3 (p. 27).

This running example could be roughly transformed into a knowledge library as follows. We slice the model from figure 2.3 (p. 27) into three models, one for `Passenger`, one for `Airport`, and one for `Vehicle` and comprising all adjacent elements, i.e., even the enumerations. Then, we represent each model on a meta-level as a node and connect them if the respective models were connected to obtain a graph similar to figure 3.4. At this point, we have lost the relationships between the new models, but we come back to this later.

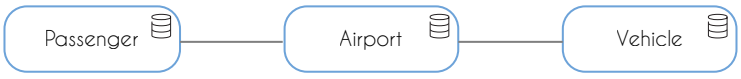


Figure 3.4.: Models Graph Example for Running Example from figure 2.3 (p. 27)

In fact, this only sketches the idea of a knowledge library, but the meta-level and the model-level appear quite clearly. Later, we will come back to this and explain when to consider a model as a white box and when as a black box.

With the sketch above in mind, we propose a knowledge library based on our previous work and show how models can be related to support model reuse [GL13; TGL13]. Moreover, we discuss issues of indexing and querying in our knowledge library. In addition, we provide extension points that foster model evolution, as proposed in section 3.4, and lay the foundations for model reuse, as explained in section 3.5.

3.2.1. Model Data Framework

On a conceptual level, our persistence comprises a central component, which we call the *model data framework*, surrounded by four extending parts, as depicted in figure 3.5. The extending parts are well-known approaches tailored to different extents. First, the *versioning* mechanism applied so far is a standard source code version control mechanism without conceptual alterations. Second, the *indexing* designed in subsection 3.2.2 uses well-known concepts tailored to our knowledge library for the purpose of insertion, deletion, and retrieval. The retrieval goes beyond regular *querying*, as we show in subsection 3.2.3, and builds on properties of models, or features in the context of recommender systems, e.g., their inherent graph structure for structural queries. In addition, a *User Interface (UI)*, only mentioned for completeness, takes over editing support for the functionality offered by the data framework.

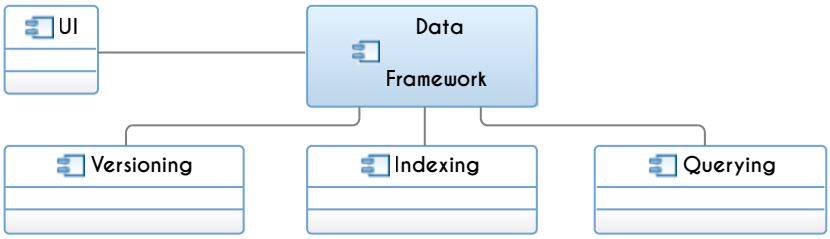


Figure 3.5.: Model Data Framework (MDF)

Subsequently, we describe the data framework from figure 3.5 and explain the meta-structure it represents. Therefore, we discuss the meta-information provided for every model stored and examine how models can be categorized, grouped, related, and interlinked, as summarized in figure 3.6. Altogether, this meta-information builds the foundation for our knowledge library as we engineered it [GL13].

Disambiguation: The term knowledge library, which is an alias for enhanced models graph library (cf. table 3.4), is the result of naming the sum of its parts, as we show subsequently. Starting with models, we can add meta-information to obtain enhanced models because of the additional information for models. Providing links instead, we obtain a models graph, because we can regard models as vertices and links as edges in a graph. If we add indexes to a set of models, we obtain a models library because of the added querying possibilities. Table 3.4 lists all the possible variations and shows that an enhanced models graph library is the fully featured result. For convenience, we introduced the term knowledge library as an alias. Note that we do not claim that an enhanced models graph is a representation of knowledge.

Table 3.4.: Terminology: Types of Data Sources for MDF

	Models	Meta Info	Links	Indexes
Models	✓			
Enhanced Models	✓	✓		
Models Graph	✓		✓	
Enhanced Models Graph	✓	✓	✓	
Models Library	✓			✓
Enhanced Models Library	✓	✓		✓
Models Graph Library	✓		✓	✓
Knowledge Library[1]	✓	✓	✓	✓

[1] Alias for Enhanced Models Graph Library

Concepts for Representing Models

Our example in figure 3.4 laid out how our running example from figure 2.3 (p. 27) could form a knowledge library, granted meta-information and an index are provided. This means that each node in figure 3.4 represents an extract from the running example as a model as well as respective meta-information. For example, the `Airport` model, which is in the middle of figure 3.4, summarizes the relevant elements from figure 2.3 (p. 27) and respective meta-information as we explain below.

For such elements, our knowledge library provides the `LibraryElement` element on the topmost level (cf. figure 3.6). This is meant to represent the reusable artifacts, which will eventually be provided by the knowledge library. Therefore, it is minimal with respect to its interface, because the actual reusable artifacts could be of different types, e.g., `LibraryElements` could be models as shown, parts of Domain Specific Language (DSL) grammars, such as in Xtext [EB10], or source code snippets, such as in Java or a DSL. For the sake of simplicity, we do not include these extensions in the concept presented here. The essential point about `LibraryElements` is that they comprise a compulsory name and list of files, as well as an optional owner.

The list of files needs to hold at least one element as a reference. We opted for a list of file references, because some tools split model information in different files. For example, one file could hold the pure data, another the representation details, and yet another the underlying grammar. Moreover, the referential nature of the file list enables distribution in several respects. A model provider, which could be the local file system, a regular web server, a database server, or a version control system, could hold the model files. However, even a mixture of these would not break a properly set up remote indexing engine or graph database, as we explain later. Further, in some of our realizations, we add a screenshot of the representation, because rendering them during runtime can cause a considerable delay. For example, one of our tools that produces previews similar

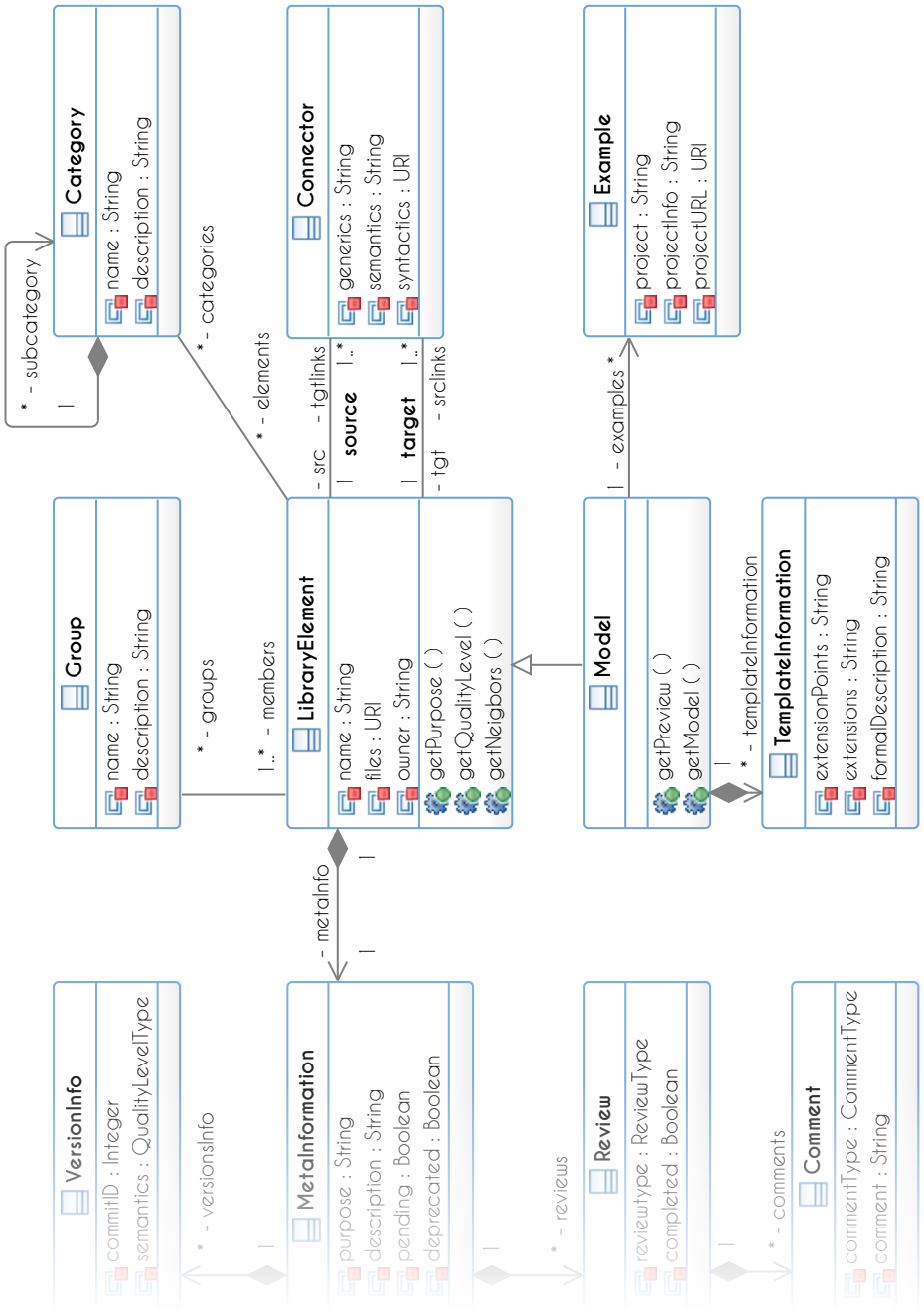


Figure 3.6.: MDF: Knowledge Library (similar to [GL13] complete diagram: page 200)

see figure 3.30 (p. 111) for faded part

to our vision in figure 1.5 (p. 8) presents screenshots. A side effect of saved screenshots is that they can provide highlights, alterations, or annotations. This enriches the preview with additional information about the previewed model.

Note that some behavior is specific to `Models` and not available for `LibraryElements`. The abovementioned previews are an example, because alternative implementations for dealing with the actual model platform exist. Common platforms include EcoreTools, Papyrus, and Sirius. Additionally, the `Model` elements provide the effective functionality for dealing with models in detail, i.e., providing them and their content. This could be a file reference, a file descriptor, or a `Resource`, in the case of EMF. Moreover, `Examples` are specific extensions to `Models`, as they are scoped accordingly with project information (cf. figure 3.6).

Another extension, this time for `LibraryElements`, is meant to enhance the information available about `LibraryElements`. Hence, we introduce a `MetaInformation` element that serves as a container for further extension regarding quality, as we will see in section 3.4, and contains extra information about the related `LibraryElement` on a meta-level we call `Purpose`. This is a lightweight specification mechanism meant to foster quality assurance in section 3.4.

For more general `Models`, we created extensions called `TemplateInformation`, as depicted in figure 3.6. These support modelers while reusing the `Model` (or rather instantiating it). Thus, if a modeler wishes to add a façade [Gam+95], the potential components of the subsystem are offered. Alternatively, if a modeler would like to add a decorator to a component [Gam+95], the component to be decorated can be chosen from a list of existing elements, and relationships and subclasses are created automatically. Altogether, this mechanism offers guided completion for textual or structural elements by linking variables and providing extension points.

Figure 3.6 depicts faded elements, which we have already alluded to. These are extensions to our knowledge library in respect of evolution and quality assurance. We postpone explaining these parts, and come back to them when we discuss evolutionary aspects in section 3.4. The other parts in figure 3.6 are introduced for the purpose of grouping, categorizing, and interlinking `LibraryElements`, as we now demonstrate.

Concepts for Interlinking and Organizing Models

The `LibraryElements` introduced above are merely a starting point, and represent no more than isolated `Models`. This means that they only represent the nodes in figure 3.4, and, consequently, the relationships between nodes are not yet represented. However, a knowledge library requires links between `Models`, and we introduce a concept for that below. Moreover, we add two mechanisms for organizing `LibraryElements`.

The interlinking elements introduced for `LibraryElements` are of two different kinds. The first comprises indirect associations via categorization and grouping (cf. figure 3.6). The former uses `Category` elements to build a set of related elements that fall into one category, but might be otherwise unrelated. For example, our `Airport` model extract from figure 3.4 could fall under a category `Transportation Hubs`, and other examples

for this category could be `Station` or `Harbor`, as depicted in figure 3.7. Other than that, `Groups` summarize `LibraryElements` that belong together in a purely semantic manner. This holds true for our `Airport`, `Passenger`, and `Vehicle` model extracted from figure 3.4, and a name for the summarizing group could be `Airport and Surroundings`. Note that the indirect interlinking elements could be derived from a higher-level concept, e.g., a concept `Links` could be the superconcept of `Group` and `Category`. This is even reasonable when regarding the attributes that both contain. We omit this, because we do not want to clutter our design.

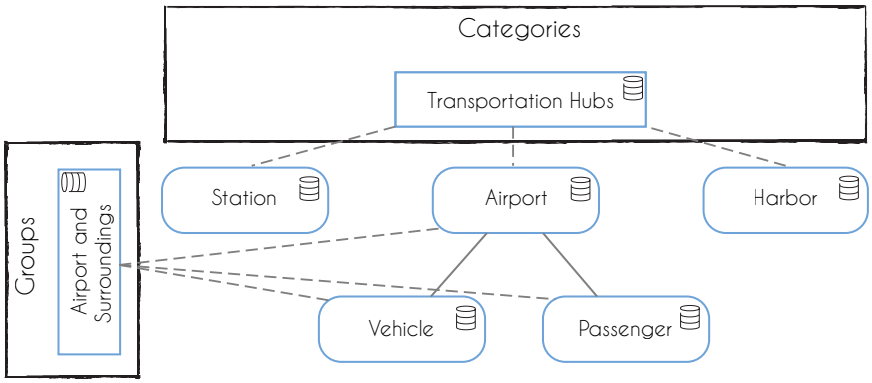


Figure 3.7.: Example: Groups and Categories related to figure 2.3 (p. 27)

The second kind of interlinking element introduced for `LibraryElements` is a direct association via `Connectors` (cf. figure 3.6). These represent that adjacent `LibraryElements` were already related in terms of their generic, semantic, and/or syntactic nature.

The generic nature of a `Connector` represents two related `LibraryElements` that are linked somehow, without providing further details. This may be because information about their relationship is not available, outdated, or deprecated. Thus, the semantics of a generic nature of a `Connector` could be read as “these two `LibraryElements` have been used together in the past”. Note that this means there is no additional, e.g., actual semantic or syntactical, information available, which implies in our example from figure 3.4 that all three model extracts could be merged back to one model, but the relationships between the classes `Passenger`, `Airport`, and `Vehicle` would be lost as well as the design rationales.

The semantic nature for `Connectors` focuses on design rationales. This means that information provided by the semantic nature is mostly documentary and can be used to aid reuse. Consider a `Patient` from figure 3.8 in the role of an adapter for a `Person` [Gam+95], with some associated `PatientID`. During reuse, a modeler might find that obvious. However, what if the adapter also prevents access to some properties of `Person` that are otherwise publicly exposed, such as `religion`? This makes our adapter a proxy as well [Gam+95]; although this is absolutely reasonable, it might not be obvious. Hence, we added the semantic nature, but still lack syntactic information if we merge models from two or more `LibraryElements`.

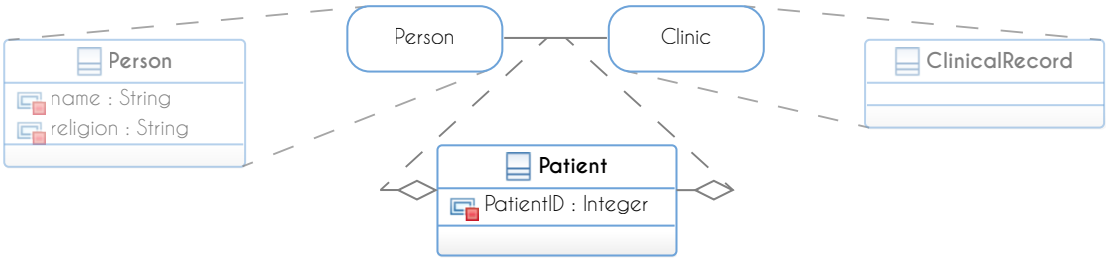


Figure 3.8.: MDF Example: Cross-link

We designed a syntactic nature for `Connectors` so that we could restore this information. Thus, merging all nodes and edges in our example from figure 3.4 would result in the original model, as depicted in figure 2.3 (p. 27). However, this feature requires additional information, which we store in an attribute of a `Connector` as an `URI`. Generally, this `URI` might point to information that provides more than just an association, as in our example. It might even contain complete bridging models with references to the adjacent `LibraryElements`. This is reasonable if a `Connector` is supposed to hold a class that plays the role of an adapter [Gam+95]. An example could be a `LibraryElement Person` and a `LibraryElement Clinic` that are interlinked via a `Connector Patient`, as illustrated in figure 3.8. Whereas the `Person` and `Clinic` model are as expected, the model in the `Patient Connector` would enhance the `Person` with additional information such as the `PatientID`. In fact, there is more to this syntactical information, like multiple involved `LibraryElements`. We call this syntactic information a cross-link and introduce it in greater detail in subsection 3.3.3.

Enhanced Models Graphs

Given the concepts in figure 3.6, we assume that `LibraryElements` and `Models` form one set of elements and `Connectors` form a second. The first set can be considered the vertices of a graph, and the second set can be considered the edges of a graph. This graph structure, let alone an index, is almost what we are aiming for, because we want to store our `LibraryElements` in an enhanced models graph library (cf. table 3.4).

However, there is slightly more to our enhanced models graph, because we need to consider `Categorys` and `Groups` as vertices, too. Given table 3.5, which corresponds to the concepts in figure 3.6, we define our set of vertices as $EMG_V := E_{le} \cup E_{Cat} \cup E_C \cup E_G$. Note that we use `LibraryElements` as a union of all derived concepts, as we did in subsection 3.1.1. Further, note that the concepts `MetaInformation`, `TemplateInformation`, and `Example` are part of neither the vertices nor the edges, because they solely provide information on a meta-level.

The edges of our enhanced models graph are defined as $EMG_E := \rho_{eSubcategory} \cup \rho_{eCategory} \cup \rho_{eConnector} \cup \rho_{eGroup}$. In doing so, we also have table 3.5 in mind. This means that we define relations as tuples of concepts, as can be derived from figure 3.6. For the sake of simplicity, we omit the names provided as well as whether the relationships are directed.

Table 3.5.: Concepts and Relationships in MDF

Concept	Set-ID
LibraryElement[1]	E_{le}
Models	E_M
Categorys	E_{Cat}
Connectors	E_C
Groups	E_G

Relation-ID		Relation
$\rho_{\text{eSubcategory}}$	$\subseteq$	$E_{Cat} \times E_{Cat}$
$\rho_{\text{eCategory}}$	$\subseteq$	$E_{le} \times E_{Cat}$
$\rho_{\text{eConnector}}$	$\subseteq$	$E_{le} \times E_C$
ρ_{eGroup}	$\subseteq$	$E_{le} \times E_G$

[1]Concepts in *italics* (and lowercase index) are abstract

Altogether, we have two notations for an enhanced models graph (EMG); first, a graph of vertices and edges, and second, a sequence of concept and relationship tuples:

$$EMG := (EMG_V, EMG_E) \tag{3.19}$$

$$EMG := (E_{le}, E_C, E_{Cat}, \ldots \rho_{\text{eConnector}}, \rho_{\text{eCategory}}, \ldots). \tag{3.20}$$

Example: We can demonstrate the notation given above by writing down our example from figure 3.4. First, we define the three vertices, which represent the model extracts from our running example in figure 2.3 (p. 27), and put them in a set $E_M^{\textit{figure 3.4}}$ as defined in table 3.6. If we want to indicate a single element, we can write $\varepsilon_M^{\texttt{Airport}}$, just as we did in subsection 3.1.2, using a superscript notation. Second, we define both required `Connectors` as follows: $E_C^{\textit{figure 3.4}}$, as given in table 3.6, where we use an abbreviation of the adjacent elements to build the names for the `Connectors`. To keep things simple, we omit the reverse direction, e.g., `Airp2Pass` and assume undirected edges for this example. Finally, we establish the given relationships from figure 3.4 and define $\rho_{\text{eConnector}}^{\textit{figure 3.4}}$ as given in table 3.6.

Table 3.6.: Sets built for model extracts from figure 3.4 (p. 48)

Set-ID		Set
$E_M^{\textit{figure 3.4}}$	=	{`Airport`, `Passenger`, `Vehicle`}
$E_C^{\textit{figure 3.4}}$	=	{`Pass2Airp`, `Airp2Vehi`}

Relation-ID		Set
$\rho_{\text{eConnector}}^{\textit{figure 3.4}}$	=	{(`Passenger`, `Pass2Airp`), (`Pass2Airp`, `Airport`), (`Airport`, `Airp2Pass`), (`Airp2Pass`, `Passenger`)}

Altogether, we obtain $EMG^{\textit{figure 3.4}} := (E_{le}^{\textit{figure 3.4}}, E_C^{\textit{figure 3.4}}, \ldots \rho_{\text{eConnector}}^{\textit{figure 3.4}}, \ldots)$, because, once more, E_{le} includes E_M. Further, if we define $EMG_V^{\textit{figure 3.4}}$ and $EMG_E^{\textit{figure 3.4}}$ as the union sets according to the above definition, we can also write down our enhanced

models graph as $\mathrm{EMG}^{\textit{figure 3.4}} := (\mathrm{EMG}_{\mathrm{V}}^{\textit{figure 3.4}}, \mathrm{EMG}_{\mathrm{E}}^{\textit{figure 3.4}})$.

Altogether, we have developed an enhanced models graph that maps to property graphs [RWE13], because we have labeled the vertices and (directed) edges. To enhance this to a knowledge library, we need to look into two topics: first, add indexing, and second, provide querying functionality. These are not trivial, and require a semi-formal notation to keep things simple.

3.2.2. Indexing Model Data

The enhanced models graph presented above provides organization and additional meta-information, but no support for finding content using a given keyword, i.e., lacks indexes (cf. table 3.4). Finding content is a task for a querying mechanism built on indexes, which could use naïve or advanced approaches depending on the strength of the indexes [Mul12]. We postpone our discussion of querying to subsection 3.2.3, but prepare the foundations for it in the form of several indexes and respective operations.

In a nutshell, we require our indexes to support queries for a given keyword and additional parameters as type information. For our example in figure 3.4, an index could comprise tuples for a dedicated kind, e.g., for the name of a `Model`. This index is then denoted as $\mathrm{I}_{\mathsf{Name}}$ and our `Airport` `Model` could be roughly represented as a tuple ("Airport", $\varepsilon_{\mathrm{M}}^{\texttt{Airport}}$), meaning that the keyword "Airport" is the `name` attribute of our `Model` $\varepsilon_{\mathrm{M}}^{\texttt{Airport}}$.

This will allow us to improve our enhanced models graph from the above to a knowledge library ($\mathrm{KL} \in \mathcal{KL}$) by simply adding the set of indexes (KL_{I}). Hence, in this case, we take into account this set and rename our vertices as well as our edges from our enhanced models graph, and obtain a knowledge library denoted by:

$$\mathrm{KL} := (\mathrm{KL}_{\mathrm{V}}, \mathrm{KL}_{\mathrm{E}}, \mathrm{KL}_{\mathrm{I}}) \quad \text{with } \mathrm{KL}_{\mathrm{V}} := \mathrm{EMG}_{\mathrm{V}}, \mathrm{KL}_{\mathrm{E}} := \mathrm{EMG}_{\mathrm{E}}, \text{ and } \mathrm{KL}_{\mathrm{I}} := (..., \mathrm{I}_{\mathsf{Name}}, ...) \tag{3.21}$$

This raises two questions: what would an indexing approach that improves our enhanced models graph to a knowledge library actually look like? Furthermore, what requirements does it need to meet?

First, the indexing is required to work seamlessly in a given environment, i.e., it needs to provide a reasonable level of abstraction and process data automatically. In terms of a later realization, this means that one method for indexing any element and one for unindexing should be sufficient. In addition, changes should affect all the indexes involved as soon as elements, e.g., `Models`, are inserted, updated, or deleted. Further, the realization should be able to determine the proper actions itself, which means it needs to detect the properties of elements to be indexed itself, e.g., whether it is of type `Model` or `Group`, and behave accordingly for each.

This leads to the need for data to be treated "right", because a particular index might not contain each and every detail of an element. In the case of indexing a model, this leads to the exclusion of data, because some properties might not be relevant. Consider two references between two classes, with one marked as the opposite of the other. Thus far, we have not considered the property of "being opposite" worth indexing. Similarly, treating

data in the “right” manner might lead to implicit data. Consider the category “Finance” ($\varepsilon_{Cat}^{\texttt{Finance}}$) and its type `Category`. This should be reflected in determining the proper index set, i.e., I_{Cat}, but not in a new keyword “category”. In addition, structural information about models should be reflected during indexing. This could be class hierarchies or delegation sequences, which enable graph-isomorphism queries. An exemplary query could be: “Find models which extend the inheritance hierarchy ‘`Creature`, `Human`’”. This query could result in a model containing `Female` and `Male` as subclasses of `Human`.

Additionally, some data we would like to see in our indexes results from plain text, which naturally inherits irregularities. Such text occurs in our `description` or `purpose`, and could contain noise words like articles, and prepositions (e.g., “the” or “to”). These should be considered as stop words and ignored [MRS08]. Words of different forms, like plurals, should be treated equally. This is commonly known as stemming [MRS08; Wil06; Por80].

However, this does not complete our indexing, which might be impacted by changes in requirements or by an altered / enhanced models graph. In other words, new requirements for queries, which might be induced by a new reuse approach, could require additional indexed information from model features. For instance, structural information should be leveraged and the length of circles is important. This would require a new index that simply addresses the lengths of circles in a `Model`. Further, new children of `LibraryElements` should not necessarily be ignored, and their treatment is likely to be different compared to `Models`, because the extraction of data will be different. Considering a new child `Snippet`, which represents parts of textual DSLs, extracting valuable data from plain text for an index certainly differs from doing so using an XMI document.

Table 3.7.: Index Sets

Indexed Element	Index-ID	Indexed Property	Index-ID
`Category`	I_{Cat}	*Words Occurrence*[1]	I_{woccur}
`Group`	I_{G}	Words in `description`	I_{WDescr}
`Model` Properties	I_{MP}	Words in model (m)	I_{WModel}
Concept `names`	I_{Name}	Words in `name`	I_{WName}
Concept Types	I_{Type}	Words in `purpose`	I_{WPurp}

[1] Concepts in *italics* (and lowercase index) are abstract

We address the given requirements by providing several indexes, summarized in table 3.7, produced by the respective strategies, as illustrated in figure 3.9 [Gam+95]. All of them will be placed in a registry and eventually handled by a service (section 4.2 (p. 153)) that delegates the elements to be indexed to the appropriate strategies. We opted for this solution for reasons of “separation of concerns”, and because some indexes consider more than one property at a time. This could scatter calls to indexes across all setters and degenerate our solution, as well as increase the risk of lost updates in the case of

careless realizations. In more detail, our indexing service forwards a new, changed, or deleted instance of a concept from figure 3.6 to all registered `IndexingStrategies`. An overview is illustrated in figure 3.9, and we will elaborate on these strategies subsequently.

The topmost concept is the `IndexingStrategy`, which denotes a common representative for all subsequent `Indexers`. Basically, this provides `canHandle(Vertex)` and `getIndexMap(Vertex)`, both of which are required for further processing by the service, as we explain in section 4.2 (p. 153). We postpone the implementation details of why the methods take parameters of type `Vertex`.

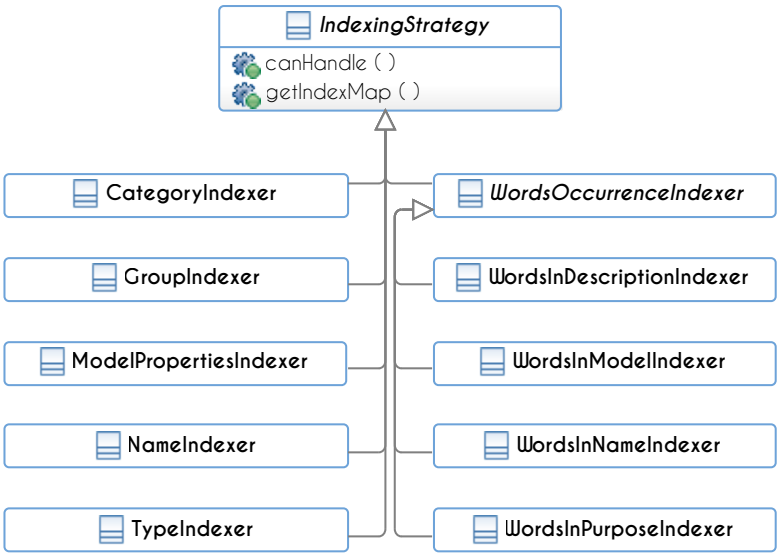


Figure 3.9.: MDF Indexing

The `Indexer`, as summarized in figure 3.9, can be subdivided into word occurrences and other `Indexers`. The `WordsOccurrenceIndexer` itself is a superconcept summarizing the `Indexer` according to the `WordsIn<`*property*`>Indexer` naming convention. This means that all these `Indexers` extract words from the <*property*> to work on. They all base their data processing on the stemming and stop words removal provided by the `WordsOccurrenceIndexer`. As an example, the `WordsInDescriptionIndexer` parses the `description` attribute of a `Group` element, as mentioned above, and provides a list of words to be indexed along with this `Group` element. Note that this `Indexer` is built to work on other elements containing a `description` attribute as well, e.g., `Category` elements or `LibraryElement` elements. Other types of `WordsOccurrenceIndexer` treat `Model` elements, i.e., extract words from models or process `name` attributes, e.g., split and parse camel-cased text in a similar manner to that mentioned above.

The `Indexer` denoted above as “other” is comprised of several supporting `Indexers`. First, the `CategoryIndexer` and `GroupIndexer` treat `Category` and `Group` elements, as expected. They roughly associate their name with the given element. Similarly, the `NameIndexer` treats `name` attributes as they are in a separate index. Note that this is different to the `WordsInNameindexer`, which does some extra processing. Next, the

`TypeIndexer` relates a given element to its type, which is similar to "typing by adjacency". For example, a `Group` element is associated with a keyword identifying it, e.g., simply "group". Note that this `TypeIndexer` must not exclude any correct given element and that `Connector` elements are also treated. Finally, the `ModelPropertiesIndexer` associates calculated or derived properties to the given element. These might be the number of classes, depth of inheritance paths, and so on. These model properties are of vital importance in our similarity searches and model isomorphism in subsection 3.2.3.

Overall, each `IndexerStrategy` provides direct or indirect (derived) data to the set of indexes ($\mathbb{KL}_I$). For example, the `CategoryIndexer` provides an index denoted as I_{Cat} with data taken from an element, whereas the `WordsInModelIndexer` contributes to the index named I_{WModel} with words taken from the model after omitting stop words. Table 3.7 contains all indexes and a short description; the subsequent list provides a more detailed overview. Further, we introduce a set of index identifiers (I_{IDs}). To summarize, we gain a set of indexes representing the direct and indirect (derived) properties of our concept, and denote it as:

$$I_{IDs} := \{\text{Cat, G, MP, Name, Type, WDesc, WModel, WName, WPurp}\} \tag{3.22}$$

$$\mathbb{KL}_I := (I_{Cat}, I_G, I_{MP}, I_{Name}, I_{Type}, I_{WDescr}, I_{WModel}, I_{WName}, I_{WPurp}) \quad \text{with details:} \tag{3.23}$$

I_{Cat} tuples of `name` attributes and `Category` elements

I_G tuples of `name` attributes and `Group` elements

I_{MP} tuples of several model properties of `Models`, e.g., number of classes, and `Model` elements

I_{Name} tuples of `name` properties of a concept and, if available, related elements, e.g., for `Group` or `LibraryElement`

I_{Type} tuples of concept types and related elements (typing mechanism)

I_{WDescr} tuples of words from `descriptions` extracted, e.g., from `Group` or `Category` elements, and related elements

I_{WModel} tuples of words from models of `Models` and related elements (e.g., words that are class names, attributes)

I_{WName} tuples of words extracted from `name` attributes, e.g., from `LibraryElements` or `Groups`, and related elements

I_{WPurp} tuples of words from `purpose` attributes related to `LibraryElements` and related elements

To provide a notation, we introduce an operation for indexing (I): $I(\mathbb{KL}, \text{element})$. The parameters are a knowledge library ($\mathbb{KL}$) and any element from figure 3.6 to be indexed. The indexing is then processed according to the explanations above. An abbreviated notation, similar to equation (3.14) (p. 39) with the subscript providing the scope and the superscript taking the element, is:

$$I^{\varepsilon}_{\mathbb{KL}} := I(\mathbb{KL}, \varepsilon), \quad \text{with } \varepsilon \in \mathbb{KL}_V \tag{3.24}$$

Example: As an example for our defined indexes, we can combine our running example from figure 2.3 and our group/category example from figure 3.7. This results in the following `Model`, `Category`, and `Group` elements:

$$\mathsf{E}_{\mathsf{M}}^{\textit{figure 3.7}} := \{\varepsilon_{\mathsf{M}}^{\texttt{Airport}}, \varepsilon_{\mathsf{M}}^{\texttt{Harbor}}, \varepsilon_{\mathsf{M}}^{\texttt{Passenger}}, \varepsilon_{\mathsf{M}}^{\texttt{Station}}, \varepsilon_{\mathsf{M}}^{\texttt{Vehicle}}\} \tag{3.25}$$

$$\mathsf{E}_{\mathsf{Cat}}^{\textit{figure 3.7}} := \{\varepsilon_{\mathsf{Cat}}^{\texttt{TransHubs}}\} \tag{3.26}$$

$$\mathsf{E}_{\mathsf{G}}^{\textit{figure 3.7}} := \{\varepsilon_{\mathsf{G}}^{\texttt{Airp\&Surr}}\} \tag{3.27}$$

To keep things comprehensible, this example is rather incomplete. We treat the `Harbor` and the `Station` models as empty and make the following assumptions: (i) The superscript is the `name` attribute and its expansion the `description`, e.g., "`Airport` and `Surroundings`" for `Airp&Surr`. Hence, our `Model` elements have no `description` or `purpose`, but we demonstrate the related mechanism with the `Category` and `Group` elements provided, at least for the `description`. (ii) Moreover, we provide only exemplary information for the calculated properties in I_{MP}, e.g., depth of inheritance tree (DIT) [CK94]. (iii) Finally, note the alternative formulation for the tuples with an extra identifier:

$$\mathsf{I}_{\mathsf{Cat}}^{\textit{figure 3.7}} = \{(\mathsf{TransHubs}, \varepsilon_{\mathsf{Cat}}^{\texttt{TransHubs}})\} \stackrel{\text{or}}{=} \{(\mathsf{Cat}, \mathsf{TransHubs}, \varepsilon_{\mathsf{Cat}}^{\texttt{TransHubs}})\} \tag{3.28}$$

$$\mathsf{I}_{\mathsf{G}}^{\textit{figure 3.7}} = \{(\mathsf{Airp\&Surr}, \varepsilon_{\mathsf{G}}^{\texttt{Airp\&Surr}})\} \stackrel{\text{or}}{=} \{(\mathsf{G}, \mathsf{Airp\&Surr}, \varepsilon_{\mathsf{G}}^{\texttt{Airp\&Surr}})\} \tag{3.29}$$

$$\begin{aligned}\mathsf{I}_{\mathsf{MP}}^{\textit{figure 3.7}} = \{&(\mathsf{dit}, 0, \varepsilon_{\mathsf{M}}^{\texttt{Airport}}), (\mathsf{dit}, 1, \varepsilon_{\mathsf{M}}^{\texttt{Passenger}}), (\mathsf{dit}, 1, \varepsilon_{\mathsf{M}}^{\texttt{Vehicle}}),\\ &(\mathsf{noClasses}, 9, \varepsilon_{\mathsf{M}}^{\texttt{Airport}}), (\mathsf{noClasses}, 7, \varepsilon_{\mathsf{M}}^{\texttt{Passenger}}), (\mathsf{noClasses}, 8, \varepsilon_{\mathsf{M}}^{\texttt{Vehicle}}),\\ &(\mathsf{noRefs}, 8, \varepsilon_{\mathsf{M}}^{\texttt{Airport}}), (\mathsf{noRefs}, 2, \varepsilon_{\mathsf{M}}^{\texttt{Passenger}}), (\mathsf{noRefs}, 2, \varepsilon_{\mathsf{M}}^{\texttt{Vehicle}})\}\end{aligned} \tag{3.30}$$

$$\begin{aligned}\mathsf{I}_{\mathsf{Name}}^{\textit{figure 3.7}} = \{&(\mathsf{Airp\&Surr}, \varepsilon_{\mathsf{G}}^{\texttt{Airp\&Surr}}), (\mathsf{Airport}, \varepsilon_{\mathsf{M}}^{\texttt{Airport}}), (\mathsf{Harbor}, \varepsilon_{\mathsf{M}}^{\texttt{Harbor}}),\\ &(\mathsf{Passenger}, \varepsilon_{\mathsf{M}}^{\texttt{Passenger}}), (\mathsf{TransHubs}, \varepsilon_{\mathsf{Cat}}^{\texttt{TransHubs}}), (\mathsf{Vehicle}, \varepsilon_{\mathsf{M}}^{\texttt{Vehicle}})\}\end{aligned} \tag{3.31}$$

$$\begin{aligned}\mathsf{I}_{\mathsf{Type}}^{\textit{figure 3.7}} = \{&(\mathsf{Group}, \varepsilon_{\mathsf{G}}^{\texttt{Airp\&Surr}}), (\mathsf{Model}, \varepsilon_{\mathsf{M}}^{\texttt{Airport}}), (\mathsf{Model}, \varepsilon_{\mathsf{M}}^{\texttt{Harbor}}), (\mathsf{Model}, \varepsilon_{\mathsf{M}}^{\texttt{Passenger}}),\\ &(\mathsf{Model}, \varepsilon_{\mathsf{M}}^{\texttt{Station}}), (\mathsf{Category}, \varepsilon_{\mathsf{Cat}}^{\texttt{TransHubs}}), (\mathsf{Model}, \varepsilon_{\mathsf{M}}^{\texttt{Vehicle}})\} \stackrel{\text{or}}{=} \{(\ldots\end{aligned} \tag{3.32}$$

$$\begin{aligned}\mathsf{I}_{\mathsf{WDescr}}^{\textit{figure 3.7}} = \{&(\mathsf{Airport}, \varepsilon_{\mathsf{G}}^{\texttt{Airp\&Surr}}), (\mathsf{Hub}, \varepsilon_{\mathsf{Cat}}^{\texttt{TransHubs}}), (\mathsf{Transportation}, \varepsilon_{\mathsf{Cat}}^{\texttt{TransHubs}}),\\ &(\mathsf{Surrounding}, \varepsilon_{\mathsf{G}}^{\texttt{Airp\&Surr}})\} \stackrel{\text{or}}{=} \{(\mathsf{WDesc}, \mathsf{Airport}, \varepsilon_{\mathsf{G}}^{\texttt{Airp\&Surr}}), \ldots\end{aligned} \tag{3.33}$$

$$\begin{aligned}\mathsf{I}_{\mathsf{WModel}}^{\textit{figure 3.7}} = \{&(\mathsf{Airport}, \varepsilon_{\mathsf{M}}^{\texttt{Airport}}), (\mathsf{ATL}, \varepsilon_{\mathsf{M}}^{\texttt{Airport}}), (\mathsf{Bus}, \varepsilon_{\mathsf{M}}^{\texttt{Vehicle}}), (\mathsf{Business}, \varepsilon_{\mathsf{M}}^{\texttt{Passenger}}),\\ &(\mathsf{Break}, \varepsilon_{\mathsf{M}}^{\texttt{Vehicle}}), (\mathsf{Car}, \varepsilon_{\mathsf{M}}^{\texttt{Vehicle}}), \ldots \text{ see figure 2.3}\} \stackrel{\text{or}}{=} \{(\mathsf{WModel}, \ldots\end{aligned} \tag{3.34}$$

$$\begin{aligned}\mathsf{I}_{\mathsf{WName}}^{\textit{figure 3.7}} = \{&(\mathsf{Airp}, \varepsilon_{\mathsf{G}}^{\texttt{Airp\&Surr}}), (\mathsf{Airport}, \varepsilon_{\mathsf{M}}^{\texttt{Airport}}), (\mathsf{Harbor}, \varepsilon_{\mathsf{M}}^{\texttt{Harbor}}), (\mathsf{Hub}, \varepsilon_{\mathsf{Cat}}^{\texttt{TransHubs}}),\\ &(\mathsf{Passenger}, \varepsilon_{\mathsf{M}}^{\texttt{Passenger}}), (\mathsf{Trans}, \varepsilon_{\mathsf{Cat}}^{\texttt{TransHubs}}), (\mathsf{Surr}, \varepsilon_{\mathsf{G}}^{\texttt{Airp\&Surr}}),\\ &(\mathsf{Vehicle}, \varepsilon_{\mathsf{M}}^{\texttt{Vehicle}})\} \stackrel{\text{or}}{=} \{(\mathsf{WName}, \mathsf{Airp}, \varepsilon_{\mathsf{G}}^{\texttt{Airp\&Surr}}), \ldots\end{aligned} \tag{3.35}$$

$$\mathsf{I}_{\mathsf{WPurp}}^{\textit{figure 3.7}} = \{\varnothing\} \tag{3.36}$$

Note how, in $\mathsf{I}_{\mathsf{WDescr}}^{\textit{figure 3.7}}$, "Airport and Surroundings" became "Airport, Surrounding" with the

"and" and the plural removed. Similarly, notice how `BusinessTraveler` was omitted in $|^{\textit{figure 3.7}}_{\text{WModel}}$, because it is split into `Business` and `Traveler` (`Business` is also a literal of the enumeration `TravelClass`). In $|^{\textit{figure 3.7}}_{\text{WName}}$, the "Airp&Surr" was also split to become "Airp, Surr". In contrast, the $|^{\textit{figure 3.7}}_{\text{Name}}$ took the `name` attributes as they are. Finally, notice how typing in $|^{\textit{figure 3.7}}_{\text{Type}}$ and $|^{\textit{figure 3.7}}_{\text{MP}}$ requires no index identifier to enable flattening, because the identifiers are already predetermined by the model property. Overall, the set of available index identifiers ($|_{\text{IDs}}$) is as follows:

$$|^{\textit{figure 3.7}}_{\text{IDs}} := \{\text{Cat, dit, G, Name, noClasses, noRefs, Type, WDesc, WMmodel, WName, WPurp}\} \tag{3.37}$$

Altogether, we have an example knowledge library from combining figures 2.3 and 3.7 as follows: $\mathbb{KL}^{\textit{figure 3.7}}$ with the details from equation (3.38) and table 3.6 ($\mathbb{E}_{\mathbb{C}}^{\textit{figure 3.4}}, \rho^{\textit{figure 3.4}}_{\text{eConnector}}$).

$$\begin{aligned}\mathbb{KL}^{\textit{figure 3.7}} := (&\mathbb{KL}_{\mathbb{V}}^{\textit{figure 3.7}}, \mathbb{KL}_{\mathbb{E}}^{\textit{figure 3.7}}, \mathbb{KL}_{\mathbb{I}}^{\textit{figure 3.7}}), \text{ with} \\ &\mathbb{KL}_{\mathbb{V}}^{\textit{figure 3.7}} = \mathbb{E}_{\mathbb{M}}^{\textit{figure 3.7}} \cup \mathbb{E}_{\mathbb{C}at}^{\textit{figure 3.7}} \cup \mathbb{E}_{\mathbb{G}}^{\textit{figure 3.7}}, \cup \mathbb{E}_{\mathbb{C}}^{\textit{figure 3.4}} \\ &\mathbb{KL}_{\mathbb{E}}^{\textit{figure 3.7}} = \rho^{\textit{figure 3.4}}_{\text{eConnector}} \text{ from table 3.6, and} \\ &\mathbb{KL}_{\mathbb{I}}^{\textit{figure 3.7}}) = (|^{\textit{figure 3.7}}_{\text{Cat}}, \ldots |^{\textit{figure 3.7}}_{\text{WPurp}})\end{aligned} \tag{3.38}$$

3.2.3. Querying Model Data

The knowledge library ($\mathbb{KL}$) introduced above comprises vertices ($\mathbb{KL}_{\mathbb{V}}$), edges ($\mathbb{KL}_{\mathbb{E}}$), and indexes ($\mathbb{KL}_{\mathbb{I}}$), which altogether build the foundation to query for information. For now, we discuss the concept of leveraging the indexed information, as depicted in figure 3.10. We will not discuss realization details, but the composite structure in figure 3.10 already implies that we can think of a combination of a builder and factory pattern for constructing compound queries [Gam+95].

We subdivide `Querys` into composing, non-structural, and structural `Querys`. The first type comprises operations in an expression logic sense, and we offer conjunctive, disjunctive, and not operators. These are binary or unary, as expected, take other `Querys` as parameters, and can be used to build more complex, in our terms, `CompoundQuerys`. The structure in figure 3.10 already signals that our realization will employ a composite pattern [Gam+95]. The second and third types, i.e., the non-structural and structural `Querys`, make use of data that has actually been indexed, such as words extracted from content and countable properties extracted from structural information, as we now show.

The non-structural `Querys` depicted in figure 3.10 comprise simple and complex `Querys`. The simplest `Query` is a `TypeQuery`, which takes a type from figure 3.6 and returns all of the matching elements. In doing so, we obtain all `Group` elements from $|^{\textit{figure 3.7}}_{\text{Type}}$ in equation (3.32), i.e., $\varepsilon_{\mathbb{G}}^{\texttt{Airp\&Surr}}$. Note that some `TypeQuerys` might result in a large number of results. This holds true for some `ModelPropertyQuerys`, because the values for the

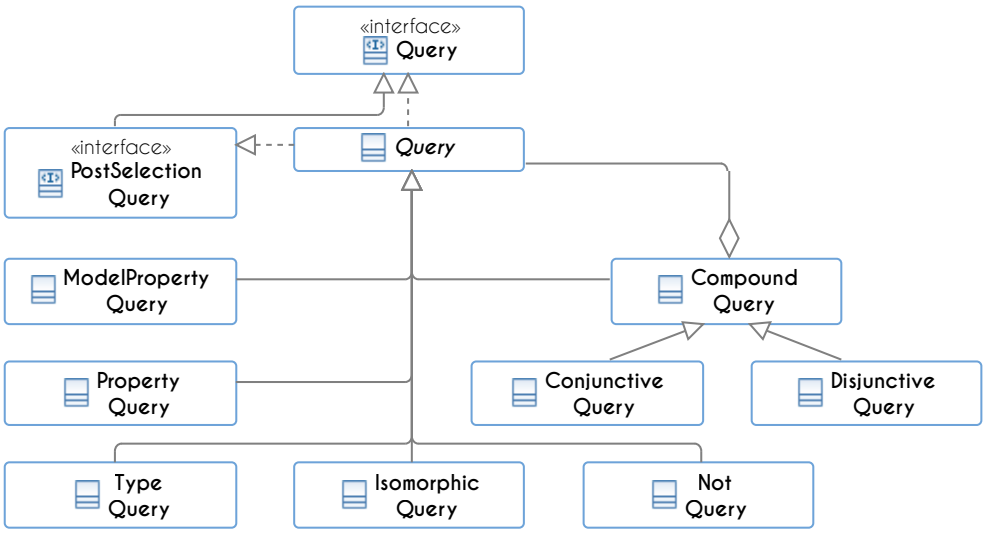


Figure 3.10.: MDF Querying

"depth of inheritance tree" range in small single-digit numbers . We counter this with a selection mechanism that will be explained later. A more complicated `Query` is a `ModelPropertyQuery`, which can leverage I_{MP} given a model property as a "keyword". For instance, if we need a model with exactly two references, our $\mathsf{I}_{\mathsf{MP}}^{\textit{figure 3.7}}$ (cf. equation (3.30)) provides $\varepsilon_{\mathsf{M}}^{\texttt{Passenger}}$ and $\varepsilon_{\mathsf{M}}^{\texttt{Vehicle}}$. Certainly, this makes more sense if we use this type of query in combination with other `Querys`, as we later show for structural queries. Finally, the `PropertyQuerys` access multiple indexes looking for queried information using the triples we introduced as alternative formulations. This indicates how the realization works by offering a set of property indicators (cf. section 4.2 (p. 153)). In detail, we can query $\mathsf{I}_{\mathsf{WModel}}^{\textit{figure 3.7}}$ (cf. equation (3.34)) in a flattened index comprising triples given by "WModel" and "Break" to get $\varepsilon_{\mathsf{M}}^{\texttt{Vehicle}}$ (later QUERYTERM:= ":wmodel Break").

With this functionality in mind, we can semi-formally define a querying operation built on `PropertyQuerys`. This takes a knowledge library, an index identifier, and a QUERYTERM as input and provides a set of vertices from the knowledge base. We denote a `Query` to our knowledge library with the uppercase (Φ) of our find symbol for elements in Ecore models (φ) from equation (3.11) (p. 38). Additionally, Ψ maps an index to its identifier. This mapping omits the identifiers for model properties (I_{MP}), because they are solely used by structural `Querys`, as we show later.

$$\Psi := \mathsf{KL}_{\mathsf{I}} \backslash \mathsf{I}_{\mathsf{MP}} \rightarrow \mathsf{I}_{\mathsf{IDs}} : \tag{3.39}$$
$$\varepsilon \mapsto \psi, \psi \in \mathsf{I}_{\mathsf{IDs}} \backslash \{\text{dit, noClass, noRefs}\}$$
$$\Phi := \mathsf{KL} \times \mathsf{I}_{\mathsf{IDs}} \times \textsc{str} \rightarrow \mathsf{KL}_{\mathsf{V}} : \tag{3.40}$$
$$(\mathsf{KL}, \psi, \textsc{queryterm}) \mapsto \{(\psi, x, \varepsilon)\}, x = \textsc{queryterm}, \varepsilon \in \mathsf{KL}_{\mathsf{V}}$$

Once again, we introduce a shortcut notation for this querying operation. Similar to

equation (3.14) (p. 39), the subscript provides the scope, which is the knowledge library, and the superscript holds the actual QUERYTERM. For a given $\mathbb{KL}$, we can write:

$$\Phi_{\mathbb{KL}}^{\text{QUERYTERM}} := \Phi(\mathbb{KL}, \mathbb{I}_{\text{Type}}, \texttt{Group}), \quad \text{QUERYTERM} = (\Psi(\mathbb{I}_{\text{Type}}), \texttt{Group}) \text{ or ":Type Group"} \tag{3.41}$$

This allows us to rewrite our exemplary queries from above for `Group` and "Break":

$$\Phi_{\mathbb{KL}^{\textit{figure 3.7}}}^{\text{QUERYTERM}} := \Phi(\mathbb{KL}^{\textit{figure 3.7}}, \mathbb{I}_{\text{Type}}^{\textit{figure 3.7}}, \texttt{Group}), \quad \text{QUERYTERM} = (\Psi(\mathbb{I}_{\text{Type}}^{\textit{figure 3.7}}), \text{Group}) \tag{3.42}$$

$$\Phi_{\mathbb{KL}^{\textit{figure 3.7}}}^{\text{QUERYTERM}} := \Phi(\mathbb{KL}^{\textit{figure 3.7}}, \mathbb{I}_{\text{MP}}^{\textit{figure 3.7}}, \text{"Break"}), \quad \text{QUERYTERM} = (\Psi(\mathbb{I}_{\text{MP}}^{\textit{figure 3.7}}), \text{Break}) \tag{3.43}$$

As a note, each `CompoundQuery` can be split up into separate `Querys` as far as possible, and then evaluated individually and combined again using set operators. This excludes `NotQuerys`, unions of `DisjunctiveQuerys`, and so on. For example, the `Query` "AND (:noRefs 2) NOT (:name 'Vehicle')" evaluates the results in $\varepsilon_{\mathbb{M}}^{\texttt{Passenger}}$.

Structural `Querys` are `IsomorphicQuerys` in our approach, and can take models as inputs. They transform these models into simple dependency graphs (cf. pseudocode 3.2 (p. 77)), which are congruent to class diagrams, and process them in two steps. First, `ModelProperyQuerys` generate a rough list of models candidates that fulfill at least the basic properties of the given models. The (partial) graph isomorphism algorithm is then executed with the given candidates to determine whether the given graph is an isomorphic graph or subgraph. This is a good example for a `PostSelectionQuery`, because it operates after candidates have already been found. Regarding the actual algorithm, we could use random walks for approximate graph matchings or an improved version of the VF algorithm called VF2. As we expect our graphs to be rather small, we opt for the exact version using the VF2 algorithm.

The idea behind the VF2 algorithm, as sketched in pseudocode 3.1, is to experiment with vertex mappings between two graphs. Given a mapping, the algorithm calculates candidate vertices for the current mapping, as shown in in line 4 of pseudocode 3.1. These candidates lead to derived edges (cf. line 5), which are tested as shown in line 7. If an edge helps to extend a partial isomorphism, the candidate is added to the mapping in line 9 and the algorithm is executed again (cf. line 10).

The candidate generation works as illustrated in figure 3.11, which shows two sets of vertices as hexagons, denoted as the domain and co-domain, which are mapped by a function m (our `Map<V,V> mapping` in pseudocode 3.1). Further in- and outgoing edges for both sets are shown. They represent adjacent candidates, because they can extend the existing mapping. The algorithm calculates potential candidates in three steps. First, the in- and outgoing edges are calculated and their adjacent vertices that are not in a mapping are distinguished in out-terminal sets, e.g., "a" or "z", and in-terminal sets, e.g., "b, d" or "w, y". Second, possible candidate pairs are built from the out-terminal sets by a

certain Cartesian product, e.g., for “a” and “z” [Cor+01; Cor+04]. Alternatively, if empty, the in-terminal sets are used, e.g., “b, d” and “w, x”; if these are also empty, the certain Cartesian product is built by considering the edges minus the current mapping.

The pairs of candidates are then tested as follows. All candidate edges with adjacent vertices in the current mapping are tested if they have already been mapped. In addition, an edge must be compatible in terms of other requirements, e.g., the weight. If these requirements are satisfied, the pair of candidates extends the current mapping successfully, but not necessarily ultimately, because some backtracking might occur.

```
1  boolean runVF2Algorithm(Map<V,V> mapping) {
2   if (mapping.containsAllVertices(subgraph))
3      return true;                  // mapping congruent with subGraph
4    Map<V,V> candidates = getCandidatePairs(mapping);
5    for (V superGraphCandidate : getSuperGraphVertices(candidates)) {
6      V subGraphCandidate = candidates.oposiOf(superGraphCandidate);
7      if (isFeasible(mapping,              // new (partial) solution?
8                     superGraphCandidate,subGraphCandidate)) {
9        mapping.add(superGraphCandidate,subGraphCandidate);
10       if (runVF2Algorithm(mapping))
11         return true;          // solution found for this descendant
12       mapping.remove(superGraphCandidate);// no descending solution
13     }
14   }
15   return false;              // no solution, for neither descendants
16 }
```

Pseudocode 3.1: VF2 Sketch

The aspects that require tailoring in our case are the candidate generation in in line 4 of pseudocode 3.1 and the edge testing in line 7. Both operations must consider model characteristics, i.e., our different kinds of vertices and edges. First, the candidate generation must only produce candidates comprising either `EClass` (E_C) or `EEnum` (E_E) vertices (cf. table 3.1 (p. 34)). Second, the candidate test must acknowledge that edges are either $\rho_{\text{eSuperTypes}}$ or $\rho_{\text{eReferences}}$ (table 3.2 (p. 36)). In addition, other vertices like `EAttributes` (E_A) and their respective edges ($\rho_{\text{eAttributes}}$) need to be treated transparently.

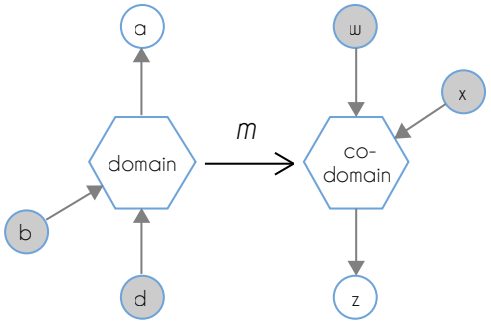


Figure 3.11.: VF2 Candidate Generation for existing mapping m

3.2.4. Design Rationales and Observations

The knowledge library introduced above addresses the “Storage” and “Representation Challenge” discussed in section 1.2 (p. 6), and in terms of Petro, Fotta, and Weisman, we build a domain-specific software architecture (DSSA) and map our requirements to concepts [PFW95]. The guide for this taxonomy for software reuse libraries is Mili, Mili, and Mittermeir, which was later refined by Mili et al. [MMM98; Mil+02]. They discuss storage and retrieval mechanism among other aspects. In this regard, they look into the properties of software libraries as the nature of assets, scope of the library, asset representation, storage structure, or navigation schema. We use this as a checklist during the development of our knowledge library, but most importantly, we attempt to counter the “obstacles to software assets classification” [Mil+02]. Therefore, we first discuss these obstacles and our conclusions with regard to our fool’s errands, design alternatives, and decisions on the enhanced models graph of our knowledge library (cf. table 3.4). We then take into account our `Querys` and `Indexer`. Finally, we contrast our knowledge library, i.e., enhanced models graph library (cf. table 3.4), with ontology.

The structure for our enhanced models graph, as presented in figure 3.6, is the result of several attempts to provide a sound foundation for a model recommender system (cf. the MoCCa second system effect in section 5.1 (p. 167) [Bro75]). Our reasons for explaining the difficulties of this encounter spring from the abovementioned “obstacles to software assets classification” [Mil+02]. Hence, we first counter the information-rich nature of our assets, i.e., models. Detaching them from their scenario of use means losing information that we could either use or try to map in our enhanced models graph. We opt for the latter and create `Categorys`, `Groups`, and `Connectors`. Further, we experiment with patterns as additional means for further structuring, but do not obtain the anticipated benefit (cf. second prototype section 5.1 (p. 167) and storing models section 6.2 (p. 189)) [Fuc11]. Similarly, so-called best practices are beyond the scope of this study. In addition, the `TemplateInformation` could potentially be used, but this is not explained in detail here (cf. storing models section 6.2 (p. 189)) [Hu13]. Overall, our approach attempts to find a middle ground that we can leverage for model recommendation production (cf. equation (3.76) (p. 120) and schema from subsection 3.5.7 (p. 138), as detailed in figure 3.35 (p. 140)). Note that this model recommendation production is only methodologically possible because of our enhanced models graph combined with indexing and querying.

The second obstacle we face is that our models may be arbitrarily similar [Mil+02]. This could lead to redundancy in our enhanced models graph. Although we do not mean to forbid that at all cost, it is important to control the degree and structure of the similarity. Hence, a distinguishing concept, our `Groups` at last resort, can help distinguish two otherwise equal models for their different usage scenarios. Put differently, they are different in our enhanced models graph because they have different neighbors. Certainly, that must be determined in the process of identifying a submodel candidate, as we discuss later (cf. section 3.3 (p. 73)).

Relating `LibraryElements` or `Models` also means that we could ease the “lack of meaningful relations between assets”, although we do not mean to provide an equivalence

or ordering relation [Mil+02]. Instead, Categorys, Groups, and Connectors provide different ways to define the distance between Models, as employed for model recommendation production (cf. equation (3.76) (p. 120) and schema from subsection 3.5.7 (p. 138), as detailed in figure 3.35 (p. 140)). Once again, this allows methodological model recommendation production and, considering distances and fallback strategies, it is possible to tailor a model recommendation production in a template format using simple configuration parameters, as sketched in appendix A (p. 195).

The rationale above raises the question if a graph is the most suitable data structure for our meta-data, or if we could use other alternatives. These could go in two directions, and either be more heavyweight or more lightweight. The former could be an ontology, and we will deal with this separately. The latter could be matrices, as used by recommender systems, or relation-centered storage such as relational databases. Further, no organization at all could be possible, as machine learning provides a means for arranging seemingly chaotic data. This suggests that we need to make a decision on a continuum from unordered meta-data to structured meta-data according to our requirements.

Unordered data on a meta-level were never an option, because we should provide at least rudimentary browsing functionality from the beginning. This means that, even for a few Models, we need a way to present them. Hence, categories are the minimum requirement to avoid brute-force navigation, and a faceted library could satisfy this need [Sch+10]. This could be realized with matrices as well.

However, matrices are not sufficient in our case, because we aim for model reuse by means of model recommendations. What happens if a model is in use already? Should we recommend this model as it is stored in our knowledge library, or generate another model from the same category? These options should only be selected if we cannot do any better. In fact, we will later deal with a threshold of completion degree, because it makes no sense to recommend a model that is already present to, say, 80%. Still, we need means to provide something beneficial, e.g., some relationship to another Model in our knowledge library. This is where our Groups come into play. However, we can persist with Categorys and Groups in separate matrices without loss of generality.

What we cannot yet do is produce so-called chain model recommendations [GL13], i.e., quick follow-up model recommendations. This is what Connectors with syntactic information are for. They provide a middle ground for separating models during harvesting and putting them together easily during reutilization. This is not possible without Groups and Connectors. In addition, and as a side benefit, the combination of Categorys, Groups, and Connectors allows for an intuitive graph-walk algorithm. For us, that means we can experiment with recommendation operations using graph query languages instead of implementing yet another algorithm.

Finally, we found a design that had minimal impact on computation times during model recommendation production (cf. subsection 3.5.2 (p. 118)). This is important, because isomorphism queries can become computationally expensive.

This raises the question of why we are not storing all information in a single graph. A so-called “Model Repository” bursts EMF models in a graph database for further processing [Eli+10]. Moreover, it recently became popular for huge models (larger than

several thousand elements) to use the EMF proxy mechanism for lazy loading from graph databases [Ben+14; Góm+15]. However, this means two things in our case. First, we would have all data for models and meta-data in one graph database, which would require further organizational effort and run counter to our desired separation of concerns for these data sources, e.g. for versioning models in evolution (cf. section 3.4 (p. 94)). Second, drawing boundaries for what to recommend would become another organizational issue. Certainly, graph walk algorithms could help, but arranging information according to its `purpose` blends nicely with our conceptual `Models` as well as with the reviews we introduce in section 3.4 (p. 94).

Other than that, the `Querys` described above will provide a toolbox for producing recommendations, and we will make use of many more `Querys` in section 3.3 (p. 73) and section 3.5 (p. 117). We will then explore the possibilities of our graph structure, which means taking into account `Categorys`, `Groups`, and `Connectors` for the purpose of graph traversal, as briefly mentioned above. We do so in the case of reuse, because a similar `Model` in our knowledge library would not contribute much to a recommender system, but neighbors of this `Model` probably would. However, this will require us to consider a detailed evaluation of the results, i.e., ranking.

Other aspects that careful readers will already have observed involve the ordering of components of `CompoundQuerys`. In particular, `ConjunctiveQuerys` allow for optimization, because the second `Query` only limits the results of the first. However, we deploy specific platforms with built-in mechanisms for optimization, e.g., elasticsearch [GT15].

Further, we attempt to develop a concept that is not only designed for a single technology, although, admittedly, our figures are UML class diagrams. This might induce a single platform, but manifold knowledge preserving approaches are possible. Certainly, this does not prove that our approach works for all methods such as ontologies, (RDF) triplestores, graph databases, or relational databases [DGD05].

Moreover, we build some redundancy into our indexes ($\mathbb{KL}_I$) for convenience, because it eases the `Query` writing. For example, `Category` elements can be found in at least two ways. Considering our indexes example from page 60, $\varepsilon_{\text{Cat}}^{\texttt{TransHubs}}$ is, first, listed in $I_{\text{Cat}}^{\textit{figure } 3.7}$ from equation (3.28), and an unlimited `PropertyQuery` returns $\varepsilon_{\text{Cat}}^{\texttt{TransHubs}}$. Second, the same element results from a `PropertyQuery` to $I_{\text{Type}}^{\textit{figure } 3.7}$ from equation (3.32) with the provided “Category”. In other words, these equivalent `Querys` look as follows:

$$\Phi_{\mathbb{KL}}^{\text{QUERYTERM1}}, \quad \text{with QUERYTERM1 = Cat} \tag{3.44}$$

$$=$$

$$\Phi_{\mathbb{KL}}^{\text{QUERYTERM2}}, \quad \text{with QUERYTERM2 = Type Category} \tag{3.45}$$

A closer look at `PropertyQuerys` shows that, given the index identifiers (I_{IDs}), all non-structural `Querys` are possible. Hence, they can be considered shortcuts, but structural and composing `Querys` are not. Yet the insight of this reaches further, because it explains that indexing requires an individual `Indexer` for properties or elements in our knowledge graph. This does not necessarily require a new kind of `Query`, let alone structural and logical types, as long the index identifier is fed to a `PropertyQuery`. Consider, for

example, an extended `ModelPropertiesIndexer` adding information about the number of children. This could lead to two (actually one) entries as follows: $(\text{noc}, 2, \varepsilon_{\mathsf{M}}^{\texttt{Passenger}})$. While this requires the metric to be implemented, the `Query` needs no more than the index identifier and a number to retrieve the information.

As we deploy our approach in reuse for conceptual modeling, it makes sense to contrast our knowledge library with ontology [Hes02], because it is believed that they could also bolster the modeling [Obe14b; Obe14a]. First, we must establish at least a rough understanding of what "ontology" means, though there is no common understanding [Bus+14]. This is because the term is used in many domains [Fur14], so we keep to computer science and limit our understanding. A more elaborate discussion covering even cognitive sciences and philosophy is provided by Stuckenschmidt [Stu09].

Gruber states that "[a]n ontology is a formal, explicit specification of a shared conceptualization" [Gru93], whereas Herre et al. considers that "an ontology is a shared understanding of some domain of interest, which is represented as a set of concepts (e.g. entities, attributes, and processes)" [Her+06]. Alternatively, [UG96] gives the following definition: "Formal Ontology is the science that is concerned with the systematic development of axiomatic theories describing forms, modes, and views of being at different levels of abstraction and granularity." Altogether, Pickert concludes that "ontologies should enhance machine to machine communication and machine to human communication" (translated from [Pic03]).

Hence, the Web Ontology Language (OWL) (abbreviated like this because it sounds nicer) provides a machine-readable format for formalizing ontology. Thus, an increasing amount of domain knowledge, e.g., genome sequences, semantic descriptions of images, laws, sentences, professional articles, is formalized in OWL. As a marginal note, this recurs in section 3.5, where we will touch on OWL DL, a decidable version that is also transformable to Ecore [ROD10].

Therefore, altogether, conceptual modeling and ontology are quite close and, regarding transformations, closely resemble one another. Generally, both suit the purpose of categorizing and relating entities [Rei83]. However, we undertake conceptual modeling for the purpose of code, or at least prototype, generation rather than knowledge deduction. This is a benefit compared to knowledge engineering [Fur14].

3.2.5. Related Work

Approaches for collections of software artifacts have been the subject of research for almost 40 years [MMM98], but comparable solutions focusing on reuse in object orientation only emerged in the 1990s. These considered either software components or prototypes. An early example of the former is presented by Goguen et al., who leverage interface information for retrieving software components [Gog+96]. We took some inspiration from their formal definitions but, for comprehensibility reasons, we remain semi-formal. Another example is that given by Macchini, who developed ESTRO to provide architectural and data flow ideas for their system [Mac92]. An example for the latter is AGORA [SHW98], which offers prototype searches based on component models.

Both solutions, though tailored for different artifacts, follow similar paths with respect to architecture and information retrieval, let alone structural queries.

For more recent work, we can take a different perspective on our persistence and distinguish our approach accordingly. Hence, we subdivide the related and complementary approaches into repositories, libraries, and other knowledge-preserving systems, e.g., ontologies. We conclude with some observations about model management.

Model Repositories: There is no agreement on the term model (cf. subsection 2.3.1 (p. 17)), nor is there a consensus on the definition of a repository (cf. subsection 2.3.1 (p. 18)). Hence, we consider the subsequent approaches as repositories, though the terms indicate otherwise.

Altmanninger et al. provide a reasoning for why research on model versioning is needed, and indicate that the solution is a repository for models providing operation- and semantic-based conflict detection [Alt+09]. Their solution is AMOR, the "adaptable model versioning repository", which overcomes the shortcomings of XMI serializations [Alt+08]. Closely related projects (and partly predecessors) are modelCVS, the "model concurrent versions system" [Kap+06], and SMoVer, the "semantic model version control system" [Alt+07; Alt08]. These repositories cohere with our understanding of the term, because they offer storage, versioning, and conflict-resolution functionality, but indexing or querying is not an issue.

France Telecom developed a UML model repository implementing the MOF stack as a central storage service [Bel99]. They store metamodels and models in this repository and have developed a suitable meta-metamodel for that the purpose. The reason for developing this repository was to unify data representation. This puts reuse of models in focus, as we do, but there are no details regarding a search functionality, though this realization would be useless if it did not support a search operation.

Other realizations of model repositories exist in the Eclipse ecosystem. The first is called EMFStore [KH10]. Developed by Kögel, this supports model evolution in a central repository through operations [Kög11]. Although there are options for finding models, the purpose does not lie in reuse, but supporting collaborative work, and hence, version control and change propagation. The second is the CDO Model Repository (connected data objects), which provides a framework for multiple backends to store and manage large models [Ecl10]. Hence, it focuses on efficiency aspects rather than reuse or querying. Finally, the "Model Repository" at the University of Leipzig is a project that uses graph databases [Eli+10]. To that end, it migrates Ecore models to graphs and offers editing functionality including merge operations. It does so by persisting classes as vertices and relationships as edges in the graph database. We mention this repository to provide a more coherent picture, although the project has been abandoned for more than five years and its precise goals are not clear.

Model Libraries: According to our understanding of the term repository, an indexing and querying mechanism can enhance it to become a library. Consequently, projects concerning indexing and querying could make EMFStore or CDO into model libraries. Examples for such projects are EMF Search (discontinued [Ecl12b]), EMFIndex (archived [Ecl12a]), EMF Query [Ecl15a], and EMF-IncQuery [Ujh+15]. These projects provide

querying and some indexing support for EMF/Ecore models. Alternatively, a single technology, e.g., Object Constraint Language (OCL), can induce querying, as proposed by Akehurst and Bordbar [AB01]. However, none of the above offers structural `Query`s.

Academia-related projects with built-in indexing and querying fostering web technologies include MOOGLE [LFW12], ReMoDD [FBC06], and MDEForge [Bas+14]. MOOGLE was first developed by Lucrédio, Fortes, and Whittle and, as the name suggests, is a web search engine for models; it is not publicly available. MOOGLE works on XMI-persisted models and extracts beneficial information for more accurate means of searching [LFW12]. The frontend, which supports fuzzy searches, ranked results, and provides previews, is designed to be user-friendly, according to the authors. ReMoDD is built and maintained at Colorado State University by France, Bieman, and Cheng, and is focused on community building and documentation [FBC06]. Hence, many models are in PDF or screen-shot format, which means they are not directly editable or machine processable. ReMoDD is often used as a platform for exchanging models and related examples, such as case studies or educational-level models. Finally, MDEForge is a community-based repository for models realized with web technologies by Basciani et al. [Bas+14]. It is meant for "development, analysis, and reuse". This summarizes the functionality of MOOGLE and ReMoDD to some extent, and should be seen as a model-management tool offering software-as-a-service.

Still in the domain of web technologies, but working on web application models, Bislimovska et al. have laid out a development process for finding reusable models across project repositories [Bis+14]. They start by presenting a conceptual architecture, which matches ours, and discuss common information retrieval mechanisms, i.e., indexing and querying words, as well as aspects of ranking results [BBF10]. They then enhance their approach using structural queries, similar to our `IsomorphicQuery`s, which they denote as "content-based queries" or "query by example" [Bis+11]. The inputs they use for querying are rather sketched ideas of what might be useful. Hence, they are more concerned with similarity than exact matches, and employ different preprocessing and algorithms than us. Hence, they first need to transform project repository content into graphs and index them properly. Then, they can use a tailored A-star algorithm for "error correcting subgraph isomorphism", enabling similarity searches for a given graph edit distance. This unveils another difference to our approach, which always works in two steps to minimize the number of candidates for the structural `Query`s.

Shao, Sun, and Chen studied the reuse of workflow models and presented the WISE search engine [SSC09]. Their approach considers workflow hierarchies that are "three-dimensional object[s] containing multi-resolution views on the same workflow". They mean to say that workflow tasks can resolve to other tasks, services, calls, and queries (multi-resolution) in their otherwise two-dimensional graph structure, i.e., tasks and subtasks. This makes indexing and querying challenging, as they demonstrate in an exemplary workflow. Compared to our approach, the proposed data processing steps, e.g., index builder, and the conceptual architecture are similar, though our focus lies on extendability.

A slightly different approach for libraries uses facets that are the result of domain

engineering and could be seen as categories. However, they are more elaborate, because a standard catalog of facets always serves as a starting point. Schmidt et al. developed a facet library for arbitrary software components and explain how facets enable more efficient development. To some extent, this is similar to categories, but actually goes beyond, albeit not as far as our `Groups`. Still, it allows for the browsing of artifacts, as we allow in our graphs.

Keeping in mind that there is a path from OWL to Ecore [ROD10], we should also look into ontology reuse. Given an ontology repository, Fernandez, Cantador, and Castells derived a method for gathering a set of terms, evaluating them with respect to the repository, and presenting a ranked list for manual reevaluation by the user [FCC06]. Note that the evaluation step corresponds to our `Query`, but evaluating an ontology is different. It might mean "comparing to a Golden Set", installing and measuring the resulting quality, comparing to unstructured data, or evaluating manually by human interaction. Additionally, collaborative filtering and other techniques from recommender systems are applied, but we do not go into further detail here and postpone our discussion to subsection 3.5.9. For now, the sequence of data processing steps in their CORE tool, including the natural language processing, is equivalent to our approach, but our ranking and evaluation produce different results.

Model Knowledge Libraries: The "Model Intelligence approach" presented by White and Schmidt is a framework for domain-specific modeling languages on a conceptual level [WS06]. They focus on establishing domain-specific knowledge bases and algorithms so they can map models in what they call "combinatorically challenging domains". For example, they semi-automatically map logical models to deployment models for car electronic control units. Hence, their modeling language is AUTomotive Open System ARchitecture (AUTOSAR) and their knowledge base employs Prolog for inference. The most notable difference to our lies in the artifacts we deal with—while White and Schmidt focus on mappings, we are concerned with models and their relationships.

Model Management: According to Bharadwaj et al. and others, the term "model management" was coined by Will [Wil75], and is not exclusive to computer science. Rather, it has been used for mathematical models in decision making, as surveyed by Bharadwaj et al. [Bha+92]. The models dealt with are intended "to maximize or minimize a set of mathematical constraints that capture the conditions under which the decisions have to be made". Therefore, several equations with constraints are solved and the solutions have a numerical nature. Hence, "[t]he goal of a model management system is to provide a modeling environment which can conceive, represent, manipulate, integrate, and control a variety of models in an organization". From our perspective, this holds true, even today, though the models under consideration in computer science have changed.

At the time, however, model management already existed in computer science, and Blanning provides, at roughly the same time as Bharadwaj et al., an overview of model management systems in computer science [Bla93]. This shows how work in both disciplines is rooted in the same foundations. Whereas decision support systems (DSS) play a major role in both texts, the latter turns toward database management. Eventually, this leads to the ideas presented by Melnik for general model management [Mel04].

More recently, a change in the terms for model management occurred, and an approach called “megamodeling” emerged [Béz+05]. This concept aims at a different application than our knowledge library, instead trying to adapt ideas from programming-in-the-large, which implements the module interconnection language (MIL) introduced by DeRemer and Kron, to modeling [DK75]. This means that modeling-in-the-large is proposed with a distinct model that plays the role of the MIL at the meta-modeling level. This distinct model stores the relationships between project assets, which are also models, and holds all of them together. This allows for traceability and enables transformations as well as weaving, which are key aspects of modeling environments according to Allilaire et al. [All+06]. Alternative formulations of issues and concepts are provided by Herrmannsdörfer and Hummel [HH10] and Krishnan and Chari [KC00] [Lev+11].

3.2.6. Summary of Storing Models

This section introduced a knowledge library (KL) in the form of concepts that build a graph structure with indexing and querying functionality. The concepts constitute three kinds of vertices (KL_V). The first kind represents the actual information, e.g., `LibraryElements`, the second kind provides meta-structures, e.g., `Categorys` and `Groups`, and the third kind emulates connections, e.g., `Connectors`. The edges (KL_E) relate the abovementioned concepts, i.e., vertices. In addition, the knowledge library comprises `MetaInformation` and an index (KL_I). This allows information to be added to the indexes (I) and retrieved by querying (Φ). For a given knowledge library, $\mathsf{KL} := (\mathsf{KL}_\mathsf{V}, \mathsf{KL}_\mathsf{E}, \mathsf{KL}_\mathsf{I})$, an ELEMENT ($\varepsilon_\mathsf{M}^{\texttt{Airport}}$) can be indexed ($\mathsf{I}_\mathsf{KL}^{\text{ELEMENT}}$) and queried for ($\Phi_\mathsf{KL}^{:\text{WMODEL AIRPORT}}$). On top of that, our knowledge library offers more sophisticated indexing and querying, including `ComposedQuerys` and `IsomorphicQuerys`. Hence, several `Querys` can be composed with basic set operations such as `DisjunctiveQuery`, `ConjunctiveQuery`, or `NotQuery`, and models can serve as both the content and structure-wise input for a `Query`.

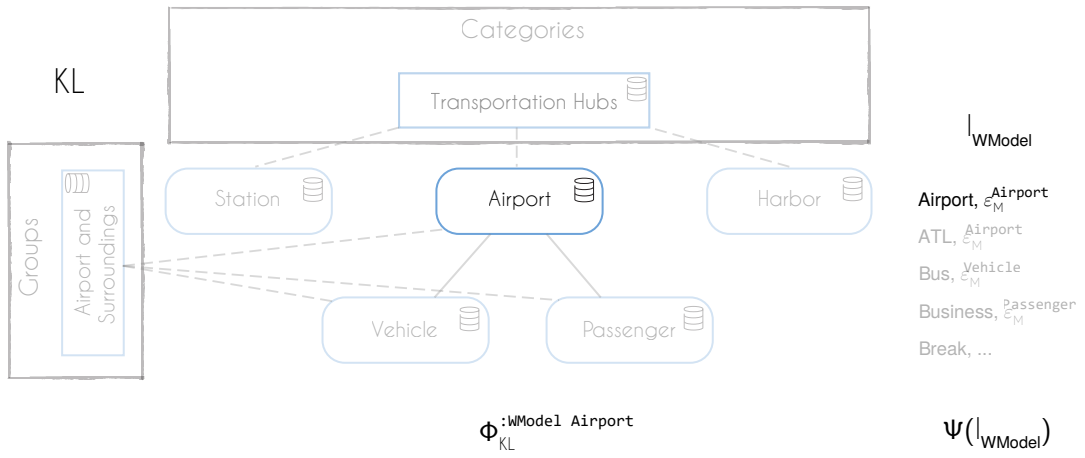
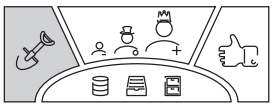


Figure 3.12.: Example Summarizing Knowledge Library (cf. figure 3.7), Index (cf. equation (3.24)), and Query (cf. equation (3.41))

3.3. Harvesting Models



Any data storage is only as useful as the data it has to offer, and our knowledge library from section 3.2 can only serve useful data if it is well fed. This feeding of data into storage is often considered a tedious task, and our knowledge library is no exception. Hence, an approach for harvesting useful data with IDEs should meet the modelers halfway, offering good integration into existing IDEs, automation, and basic quality standards for harvested models.

Of these, the most challenging requirement appears to be automation, which is related to data mining or knowledge discovery from data [HKP12]. Data mining approaches have been developed for several domains and types of data: software repositories have a long history in the mining of data for several purposes, e.g., analyzing version control systems in combination with an issue tracker for project health statements [KCM07; Hem+13]. Similarly, business processes have been identified by analyzing event data, e.g., from logging or emails, with approaches called business process mining [van+07; Aal11; AUv12]. On a more structural level for data, approaches that analyze graph data, called mining graph data, can successfully derive topological descriptors for chemical compounds, help localize software bugs, and so forth [AW12]. These approaches appear promising, because models inherit a graph structure, as we discussed in subsection 2.4.3.

Related considerations for mining graph data comprise questions of heuristics and metrics. Some metrics for models and graphs have been proposed, but processing models as graphs for harvesting sets of elements has not been extensively studied. Often, these approaches tend to dissect entire models in a predefined number of new models [Str+13], which does not help much in terms of harvesting. Thus, well-studied graph partitioning and clustering approaches for non-semantic graphs require some revision; but can algorithms produce good proposals for models?

In section 3.2 (figure 3.4 (p. 48)), we magically came up with three extracts from our running example from figure 2.3 (p. 27), but did not discuss how. An idea for how to harvest elements from models could work in three steps. First, chunks of elements could be built. Second, these could be extracted to represent new models. Finally, relationships between chunks in the original model could be extracted to represent relationships between the new models. A sketch for harvesting the `Airport`, related classes, and "adjacent relationships" from our running example may look as follows:

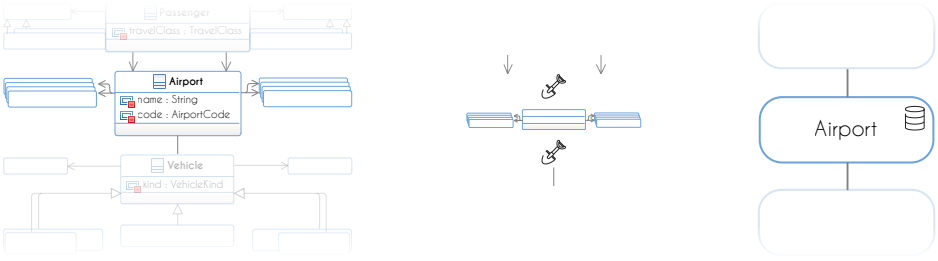


Figure 3.13.: Harvesting for the Running Example from figure 2.3 (p. 27)

This example implies that all three parts are extracted at the same time, but what would happen if the `Passenger Model` was already present in a knowledge library? Then, the steps undertaken might be as follows. First, known parts would be found and matched to a set of known parts. Second, chunks that are potentially beneficial for harvesting would be identified. Third, user interaction would insert corrections and alterations. Fourth, the finished chunks would be separated to become `Models` and build the syntactic `Connectors`. Finally, the `Models` would be stored and related via the `Connectors` with attached syntactic information, i.e., cross-links.

3.3.1. The Model Mining Framework

From the example above, we can deduce some of the components required for our *model mining framework* that is responsible for harvesting [Sew13]. Figure 3.14 provides an overview and indicates that several extensions foster harvesting. First, identifying known parts in a model under consideration might follow different approaches because of the differing intentions of modelers. Therefore, figure 3.14 shows the *marker* components intended for these different purposes. Second, one of several *splitter* is responsible for building sets of model elements, called chunks above, which are intended for reuse. Once more, the intention of a modeler determines how the sets of model elements are built. Should they be built regarding the underlying structure? Or is it worth striving for semantic dissection? Finally, the persistence is considered by *saver* components, because backends vary from directories to databases.

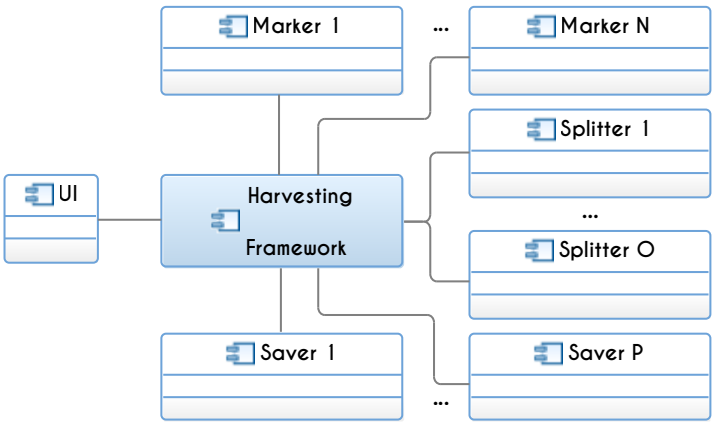


Figure 3.14.: Model Mining Framework (MMF)

We now describe how to identify beneficial parts, dissect them in new sets of elements, relate them to each other, i.e., maintain “bridging” relationships, and provide an overview of how this blends into our operation-based view of models. Note that we do not mean to fully automate this process and we have no intention of harvesting entire models, so the design of our approach is suitable for user interaction. For example, we allow users to decide on the exact dissection or degree of redundancy related to a knowledge library.

3.3.2. Identifying Submodel Candidates

Given a model (m), there are two steps involved in identifying sets of elements for reuse. First, the known elements should be identified, or marked, with respect to a given persistence unit. Though this is possible for shared directories or databases, we keep to a knowledge library. Second, the given model should be analyzed for candidates that appear beneficial for reuse, bearing in mind that known elements could lead to redundancy in a knowledge library.

The first step comprises querying a knowledge library at a given level of detail. This could take into account everything from the model, i.e., all names of classes, attributes, operations, and literals or the granularity of queries could be set to class names only. Our indexes provide options for both, but we opt for the latter for now. Hence, finding known elements involves querying a given model (m) and a knowledge library (KL). We define the set of known classes ($\mathcal{K}_{\mathsf{C}}$) as follows:

$$\mathcal{K}_{\mathsf{C}} := \{\varepsilon \in \mathsf{E}_{\mathsf{C}}^{\mathsf{m}} \ : \Phi_{\mathsf{KL}}^{\varepsilon} \neq \varnothing\} \tag{3.46}$$

Note how we use E_{C} in equation (3.46) instead of E_{ne} (cf. table 3.1 (p. 34)), thus limiting queries to classes for $\mathcal{K}_{\mathsf{C}}$. A set containing all known named elements, i.e., known elements ($\mathcal{K}_{ne}$), uses the aforementioned E_{ne} from m for querying. Another, very reasonable, set comprises all terms ($\mathcal{K}_{T}$), i.e., class names, enumerations, and relationships. The latter are important, bearing in mind that relationships are often used for modeling roles. In the second step, we analyze a given model and derive sets of elements that could be considered candidates for a knowledge library. Several formulations for this task are possible. First, it can be described as a restriction function; second, as a constraint satisfaction problem; and third, as an optimization problem. We will explain the first in greater detail and briefly provide backgrounds to options two and three later, because some internals, and possible extensions, use these formulations.

We call the abovementioned sets of elements submodels for as long as they are undergoing harvesting, and provide a semi-formal definition. Later, as soon as they are stored in a knowledge library, we refer to them as models of `Models`. Thus, given our model definition in equation (3.1) (p. 36), we can define a submodel relative to a model (m $:= (\mathsf{E}_{\mathsf{A}}, \mathsf{E}_{cl}, \mathsf{E}_{\mathsf{C}}, \ldots, \rho_{\mathsf{eAttributes}}, \rho_{\mathsf{eClassifiers}}, \ldots, \mathfrak{R})$) and denote that “s is a submodel of m” or “s is a submodel relative to m” (s $\subseteq$ m) (note tables 3.1 and 3.2 on pages 34 and 36):

$$\begin{aligned}
\mathsf{m} :=& (\mathsf{E}_{\mathsf{A}}, \mathsf{E}_{cl}, \mathsf{E}_{\mathsf{C}}, \ldots, \rho_{\mathsf{eAttributes}}, \rho_{\mathsf{eClassifiers}}, \ldots, \mathfrak{R}) && \text{given} \\
\mathsf{s} :=& (\mathsf{E}_{\mathsf{A}}|_{\mathsf{s}}, \mathsf{E}_{cl}|_{\mathsf{s}}, \mathsf{E}_{\mathsf{C}}|_{\mathsf{s}}, \ldots, \rho_{\mathsf{eAttributes}}|_{\mathsf{s}}, \rho_{\mathsf{eClassifiers}}|_{\mathsf{s}}, \ldots, \mathfrak{R}) && \text{exists, with} \\
& \mathsf{E}_{\mathsf{A}}|_{\mathsf{s}} \subseteq \mathsf{E}_{\mathsf{A}}, \mathsf{E}_{cl}|_{\mathsf{s}} \subseteq \mathsf{E}_{cl}, \mathsf{E}_{\mathsf{C}}|_{\mathsf{s}} \subseteq \mathsf{E}_{\mathsf{C}}, \ldots && \\
& \rho_{\mathsf{eAttributes}}|_{\mathsf{s}} \subseteq \rho_{\mathsf{eAttributes}}, \rho_{\mathsf{eClassifiers}}|_{\mathsf{s}} \subseteq \rho_{\mathsf{eClassifiers}}, \ldots && \text{provided}
\end{aligned} \tag{3.47}$$

$$\begin{aligned}
\mathsf{ID} &:= D \to C : x \mapsto x && \text{so that} \\
\mathsf{ID}|_{D'} &:= D' \to C : x \mapsto x \quad D' \subseteq D && \text{is a restriction.}
\end{aligned} \tag{3.48}$$

Note how the submodel (s) comprises a subset of the model sets, e.g., $E_A|_s \subseteq E_A$ for the attributes. We gain the subset by means of our restriction function (ID) in equation (3.48), which limits the domain of an identity function, so that not all of the elements that occurred in the original function get mapped to the co-domain. Other than that, the submodel (s) is a complete and sound model. We call this a submodel related to the model, but it becomes an independent model once it is added to the knowledge library.

Given the definition of a submodel in equation (3.47), we may wonder how to obtain submodel candidates that are beneficial for our knowledge library for reuse. Respecting equation (3.47), this means altering the identity function in equation (3.48), which we introduced to restrict elements. In fact, we define restriction functions for "strategies of building submodels". This means that we provide an extensible selection of possible restriction functions from which users can choose.

Graph Clustering: In terms of graph theory, our restriction functions, which we simply call algorithms for now, produce tuples of sets, building graph clusters that maximize "intra-cluster density" and "inter-cluster sparsity" [Gör+13]. Some algorithms work bottom-up, i.e., agglomerative algorithms, and some work top-down, i.e., divisive algorithms [For10]. None of them processes models immediately, but instead work on dependency graphs, which are not to be confused with model dependency graphs (MDGs) or class dependency graphs (CDGs) [LM08; LM09]. The resulting clusters are evaluated using different metrics for cluster performance, conductance, coverage, density, or modularity. Subsequently, we must keep in mind that (optimization) algorithms for graph clustering and modularity are *NP*-complete [Bra+08].

The basis for our graph clustering is the dependency graphs (DG= (V, E)). These are graph representations with model elements as vertices (V) and model relationships as edges (E). For example, classes or enumerations are represented as vertices and attributes such as inheritance or association relationships are represented as edges. The term "dependency" results from the idea that two vertices depend on each other in a relationship manner, e.g., a subclass depends on a superclass, or a composite depends on a part. Further, attributes and operation parameters establish dependencies, although they are elements in our model definition from equation (3.1) (p. 36).

Thus, given a model (m), we run pseudocode 3.2 to gain a simple or regular dependency graph. The latter is simply denoted as a dependency graph. These graphs are distinguished from each other in respect of how much information is used for building their edges. Both build vertices from classes and enumerations, but the simple dependency graph only comprises edges for inheritances and references. Hence, the simple dependency graph is congruent to a class diagram figure transformed into a graph. However, a class diagram can contain more structural information, e.g., attributes and operation parameters. Contrary to Sun, France, and Ray [SFR13], these are only transformed into edges in a dependency graph, as shown after line 15 in pseudocode 3.2.

Regarding the elements and relationships processed in pseudocode 3.2, all of them can be found in our model definition (m). This means that all calls are designed according to elements in table 3.1 (p. 34) or relationships in table 3.2 (p. 36). For example, in line 2 of pseudocode 3.2 returns exactly the `EClassifiers` defined in table 3.1 (p. 34), so we

can rewrite with E_{cl}, and later $\rho_{eSuperTypes}$, E_A, E_A, E_O, E_P.

A closer look at table 3.2 (p. 36) reveals that we did not process all relationships in pseudocode 3.2. In detail, we omitted information about exceptions ($\rho_{eExceptions}$) for the sake of simplicity and because we do not expect further semantics. The same holds for the type information contained otherwise (ρ_{eType}, $\rho_{eStrucFeatures}$). In addition, we excluded relationships to literals ($\rho_{eLiterals}$) and opposing references ($\rho_{eOpposite}$). The former are considered irrelevant for structural processing, because either an enumeration is included or not, disregarding the literals, and similarly a reference is structurally relevant or not (again, independent of the opposing relationship).

```
1  DependencyGraph createDependencyGraph(Model model, Boolean simple){
2  for (E e : model.getEClassifier()) {              // classes, enums
3      Vertex v = createVertex(e); vertices.add(v);        // vertices
4      modelMapping.add(v);                      // for reverse mapping
5    }
6    for (S s : model.getESuperTypes()) {          // inheritance edges
7      Edge e = createEdge(s); edges.add(e); modelMapping.add(e);
8    }
9    for (R r : model.getEReferences())              // reference edges
10     Edge e = createEdge(r); edges.add(e); modelMapping.add(e);
11   }
12   if (simple)
13     return buildDependencyGraph(vertices, edges);    /** DG=(V,E)*/
14
15   for (A a : model.getEAttributes()) {            // attribute edges
16     Edge e = createEdge(determineAdjacents(a)); edges.add(e);
17     modelMapping.add(e);
18   }
19   for (O o : model.getEOperations()) {            // operation edges
20     for (P p : o.getEParameters()) {                // per parameter
21       Edge e = createEdge(determineAdjacents(p)); edges.add(e);
22       modelMapping.add(e);
23     }
24   }
25   return buildDependencyGraph(vertices, edges);      /** DG=(V,E)*/
26 }
```

Pseudocode 3.2: Dependency Graph

Some of the graph clustering algorithms that will be introduced work on structural graphs only, while others process weighted graphs with additional weight information, which we have not provided in pseudocode 3.2. However, we can alter the algorithm and add a parameter, which represents the corresponding weights. We can do so for edges in lines 7, 10, 16 and 21 of pseudocode 3.2 and use weights as depicted in table 3.8 for aggregation, association, composition, and inheritance relationships [Str+14]. Further, we can add weights to vertices by altering in line 3 of pseudocode 3.2 for known elements.

Note that we sometimes need to convert a dependency graph (DG) into an adjacency

Table 3.8.: Edge Weights similar [Str+14]

Edge Type	Aggregation	Association	Composition	Inheritance
Weight	0.05	0.15	0.3	0.5

matrix. Though this simplifies the algorithm, it increases the space requirements.

Associates Clustering: Possibly the simplest approach for calculating submodel candidates applies counting and graph walking. We mark leaves in a dependency graph with values of one and sum the children's values by walking up inheritances and against associations. Then, we calculate a threshold as the square root of the number of classes and enumerations. Finally, we build submodels of all classes that exceed the threshold and all their associates. In our example, `Airport` scores 9, `Passenger` scores 16, and `Vehicle` scores 18. The threshold is about 5 ($\sqrt{27}$). As a result, we gain three submodels, as shown in figure 3.4 (p. 48). Note that this is a very naïve approach that may not always succeed, e.g., consider a triangle of classes with undirected associations. For reasons of brevity, we omit a formal definition of the restriction function, because it requires us to introduce and apply a more expressive logic for formulating tree expressions. Instead, we provide a semi-formal definition that provides a set L of tuples relating vertices and their respective levels, and an extract comprising the central classes (C) that exceed the threshold, which form the basis for associates clustering.

$$\mathsf{L} := \{(\mathsf{v}, \mathsf{l})\} : \mathsf{v} \in \mathsf{V}, \mathsf{l} = \begin{cases} 1 & \text{out-deg}(\mathsf{v}) = 0 \\ \Sigma\, \mathsf{l}_i & \text{out-deg}(\mathsf{v}) \neq 0, \mathsf{l}_i \colon (\mathsf{v}_i, \mathsf{l}_i) \in \text{succ}(\mathsf{v}), \end{cases} \tag{3.49}$$

$$\mathsf{C} := \{(\mathsf{v}, \mathsf{l})\} : (\mathsf{v}, \mathsf{l}) \in \mathsf{L}, \mathsf{l} \geqslant \sqrt{|\ \mathsf{V}\ |} \tag{3.50}$$

Girvan-Newman Clustering: Opposing the bottom-up approach described above, an algorithm employing ideas from the Girvan-Newman algorithm (pseudocode 3.3) uses a top-down, i.e., divisive, approach for clustering [NG04]. This method is normally applied to community structures, which are similar to our submodel candidates, in graphs. Hence, the algorithm processes a graph by removing edges based on "vertex betweenness" [NG04]. This is the number of all shortest paths running through this edge, and in each iteration the edge with the highest betweenness is removed. This divides a graph into smaller graphs until either single vertices remain or a termination criterion is met.

This termination criterion in line 3 of pseudocode 3.3 is one of the few adjustments we need to undertake to apply the Girvan-Newman algorithm to our dependency graph. We could consider the size of the achieved graphs or the number of graphs built from the initial graph as such a criterion. Next, we need to set the directions of the edges, which are the opposite of those indicated in a class diagram. Further, we need to treat multiple edges between vertices as "equal" for betweenness, because the algorithm is not defined for multi-graphs. Finally, the algorithm works best for graphs without anomalies, but our dependency graphs expose a structure that is similar to trees, i.e., with many leaves. These leaves are likely to be isolated by the algorithm, because the betweenness for

adjacent edges is often high, as they are frequently required for the shortest path to other vertices. For example, the classes in our `Vehicle Model`, except for the `Vehicle` class itself, score 11 in our running example from figure 2.3 (p. 27). Fortunately, the relationships that we actually want to cut off score 17, so in this example, there is no problem. However, it is reasonable to set a lower bound for the minimal number of elements in a submodel. With these adjustments in mind, we can cluster our running example from figure 2.3 (p. 27) and obtain the submodels as already known and depicted in figure 3.4 (p. 48).

```
1 Set<Model> clusterGirvanNewman(DependencyGraph dg) {
2   Matrix<V,V> bet = calculateBetweenness(dg);
3   until (shallStop(dg, bet)) {                    // bridging edges
4     Edge e = removeEdgeWithHighestBetweenness(dg, bet);
5     bet = adjustBetweenness(dg, bet, e);
6   }
7   return buildClusters(dg, bet);
8 }
```

Pseudocode 3.3: Girvan-Newman Clustering

God-Class Clustering: A trait often considered harmful in object-oriented programming inspired the name and idea for finding god-class clustered submodel candidates [Rie96; Mar01]. This approach makes use of metrics for object-oriented systems to determine essential or central elements in models, so they can be considered the centers of a submodel candidates [CK94; GPC05; SM13]. A combination of metrics can detect suspects in source code [Mar01], namely weighted methods per class (WMC) [CK94], tight class cohesion (TCC) [BK95], and access of foreign data (AOFD) [Mar01]. The latter is helpful for finding god-class candidates [Mar01], and is also applicable for UML class diagrams, whereas the others are not so well applicable in our case [GPC05]. God-class clustering counts the number of relationships relative to the total number of classifiers and ranks the top 20% as candidates. The semantic explanation is that these classes tend to summarize or provide an entry point to an area of modeling. In conceptual modeling, this is sometimes of a technical nature, because root elements ease data processing.

Given a set of god classes, we need to find out which elements to cluster into which god class. We can do so by applying a modified Floyd–Warshall algorithm. The basic version of Floyd's algorithm calculates the shortest paths in a graph [Flo62], whereas Warshall's version determines reachability [War62]. We can adjust Floyd's algorithm to deliver an additional adjacency matrix providing weights instead of distances. This enables us to separate the found god classes with respect to their most-coupled classes, i.e., traversing the heaviest edges until a threshold is reached. Floyd's algorithm works in three cascaded loops over an adjacency matrix initialized with either the weight of a connecting edge or zero. Then, it processes each pair of vertices, with one set fixed (first loop), trying to find the shortest path to the other (second loop) by optimizing the path with the previously calculated information (third loop). Note that we need to process the entire matrix because we have a weighted multi-graph. This also means that we need to deal with multiple associations between vertices. An adjustment is to add the values

from table 3.8, which requires us to convert our algorithm to a maximization problem.
Kernighan–Lin Clustering: A semi-semantic approach for clustering uses the Kernighan–Lin heuristic algorithm for graph partitioning [KL70]. This is a hill-climbing approach that works in two cascaded loops for a given initial partition, which provides balanced sets of vertices. The outer loop always starts calculating the costs for vertices with adjacent vertices in opposing sets. These are the costs of the edges bridging to other sets and the costs for remaining edges. The former are often called external costs and the latter are internal costs. Hence, we can tell whether it is beneficial for a vertex to swap sets ($d(v)=c_{ext}(v)-c_{int}(v)$). This is considered in the inner loop, which takes the maximal value of adjacent vertices establishing a bridge between the two sets ($\max(d(v)+d(w)-2c(v,w))$) with a corrective term (explained later). The two chosen vertices v and w are then placed in one set and marked as having been processed. This inner loop continues until no better solution can be found for the current edges bridging the sets or until the list of edges bridging sets has been processed, i.e., all adjacent vertices are marked. Then, the markings are released and the outer loop continues. The corrective term mentioned above is required because the bridging edge is counted twice, i.e., in d(v) and d(w), and does not contribute to a better cut because both v and w are then in the same set.

```
1  Set<Model> clusterKernighanLinHeuristic(DependencyGraph dg) {
2    Model m1, m2; buildBalancedSets(dg, m1, m2);
3    until (!improved) {
4      Set<V, int> c_ext = externalCosts(dg, m1, m2);// bridging edges
5      Set<V, int> c_int = internalCosts(dg, m1, m2);// internal edges
6      Set<V, int> d = calculateD(dg, c_ext, c_int);        // benefit
7      for (i = 1 ... nrOfBridgingEdges(dg, m1, m2)) {
8        Edge e = getMostBeneficialUnmarkedBridgingEdge(dg, m1, m2)
9        markBridgedVerticesProcessed(e);
10       swapAdjacentVertex(dg, m1, m2, e);
11       d = adjustD(dg, d, e);
12     }
13     clearMarkedVertices();
14   }
15   return new Set<Model>().add(m1).add(m2);
16 }
```

Pseudocode 3.4: Kernighan–Lin Clustering

The adjustments applied to the heuristic algorithm are as follows. First, the dependency graph (DG) we provide is a multi-graph with potentially many edges between two vertices. Hence, we need to change the corrective term to take this into account and remove double the cost of all edges relating to these two particular vertices. Second, the dependency graph should provide weights, as introduced in table 3.8, for simple dependency graphs. In addition, a regular dependency graph needs to provide weights for attributes and parameters. For now, we set this weight as equal to compositions, because their semantics are similar. Third, the heuristic algorithm must be adjusted for cuts between more than two sets. This alters the calculations for external costs, as there are more values for external costs per vertex, i.e., one for each adjacent set. This will potentially

change the runtime complexity.

As an example, figure 3.15 shows two sets with five edges to cut, where the cost for cutting like this is the sum of the edge weights. The external costs for vertex v consist of the sum for edges to the adjacent vertices v_x and v_y, and the internal costs comprise only those for the edge to the faded vertex. Further, d(v) is calculated from these values, and the same values can be calculated for $d(v_x)$ and $d(v_y)$. However, our simplified figure makes no statement about the edge weights, because table 3.8 provides an idea of how they are encoded. Assuming that v is selected to be moved to Set 2, as indicated by the black arrow, then it is cheaper to cut the edge to the faded vertex than the two to v_x and v_y in Set 2.

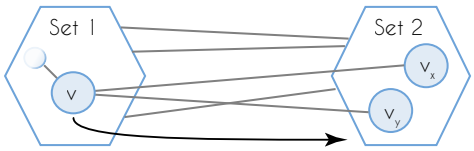


Figure 3.15.: Kernighan–Lin Vertex Exchange

Cluster Parameters: Determining a suitable number of submodel candidates for a given model (m) should be configurable and adhere to constraints that ensure comprehensibility. This means that submodels need to be understood quickly, i.e., in a matter of seconds. Hence, submodel candidates should not exceed a certain complexity, which we measure as the number of classes and enumerations ($|m|$) of a submodel or model (m), as shown in equation (3.51). Consequently, we can determine the minimum number of submodel candidates for a given threshold (c_{max}) using a function (c) defined in equation (3.52). Further, a lower bound is beneficial for some algorithms, e.g., the Girvan–Newman clustering, and we denote this minimum as (c_{min}).

$$|m| := \mathcal{M} \to \mathbb{N} : \qquad m \mapsto |E_C| + |E_E| \qquad m \in \mathcal{M} \tag{3.51}$$

$$c := \mathbb{R}^+ \to \mathbb{N} : \qquad x \mapsto \left\lceil \frac{1}{c_{max}} x \right\rceil \qquad c_{max} \in \mathbb{N} \tag{3.52}$$

Thus, algorithms are required to satisfy equation (3.53) for a given model (m) and boundaries (c_{min} and c_{max}). The equation states that the number of submodel candidates does not exceed a certain number and that the content remains within bounds with respect to the number of classes and enumerations (cf. equation (3.51)).

$$c_{min} \leqslant |s_i| \leqslant c_{max} \quad s_i \subseteq m, m, s_i \in \mathcal{M},\ i=1, \ldots n,\ n = c(|m|), c_{min}, c_{max} \in \mathbb{N} \tag{3.53}$$

Example: Let us assume that the threshold for the maximum number of classes and enumerations is 10 (c_{max} = 10). Then, equation (3.52) gives values of 2 for x = 20 and 3 for x = 21, as expected. The reason for this lies in the "ceil" function ($\lceil\rceil$), which always rounds up decimal fractions. In our first example, $c_{max}(20)$ gives $\frac{20}{10} = 2$ and $c_{max}(21)$ gives $\frac{21}{10}$, which is rounded up to 3, as desired. Note that we set this threshold to 10 for the moment, but will investigate this value further with regard to quality considerations of

models in subsection 3.4.4 (p. 101).

General Problem: Many aspects of producing submodel candidates and equation (3.53) hint at a more general problem behind graph clustering or building submodel candidates. On the one hand, during harvesting, this step can be seen as an optimization problem [Gör+13], whereas, on the other hand, it can be dealt with as a constraint satisfaction problem [LV02]. We provide an overview for the former without going into detail, as some realizations rely on these ideas. For the latter, we simply mention that the boundaries, number of submodel candidates, and strategy for distributing elements are constraints on any solution.

However, regarding the clustering approaches described above, the most obvious formulation for finding submodel candidates is an optimization problem. Our goal is to find a solution for clustering a given dependency graph (DG) while optimizing the intra-cluster density and inter-cluster sparsity. Hence, the optimization problem can be formulated as a bicriteria optimization problem that optimizes one objective while achieving a minimal quality requirement with respect to the other [Gör+13]. Often, the former is density, because cluster expansion is considered *NP*-complete.

3.3.3. Separating Submodels with Cross-Links

Thus far, we can build submodels from a given model, but cannot relate them to each other. This would induce a loss of information when the original model contains valuable information. Hence, our knowledge library provides the concept of `Connectors` to relate submodels stored as independent `Models`. In our running example from figure 2.3 (p. 27), we extracted three `Models` and some relationships were cut off. As an example, take the classes `Airport` and `Passenger`. They were related in our running example with the `source` and `destination` relationships shown in figure 3.16, but so far we have only associated the `Models` in our knowledge library. Hence, we aim to store the relationships in a `Connector`, as indicated by the dashed lines in figure 3.16.

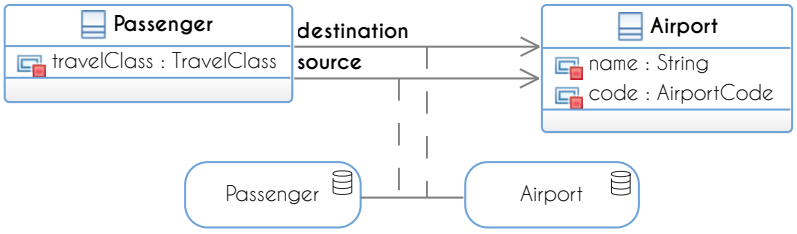


Figure 3.16.: Cross-links Example: Overview of two adjacent `Models`

This syntactical association of information between two `Models` is added to the syntactic attributes of a `Connector`, which we call cross-links [Rag11]. Figure 3.17 shows how this could be illustrated, and we define cross-links as sets for storing syntactic information. For our example, this means that the associations `source` and `destination` are put in a cross-link, as shown in figure 3.17. A cross-link (cl), for a given model (m :=

$(E_A, E_{cl}, E_C, \dots, \rho_{eAttributes}, \rho_{eClassifiers}, \dots, \mathfrak{R})$) and two disjunctive submodels (s_1 and s_2) can be written as:

$$
\begin{aligned}
s_1 &:= (E_A|_{s_1}, E_{cl}|_{s_1}, E_C|_{s_1}, \dots, \rho_{eAttributes}|_{s_1}, \rho_{eClassifiers}|_{s_1}, \dots, \mathfrak{R}) \\
s_2 &:= (E_A|_{s_2}, E_{cl}|_{s_2}, E_C|_{s_2}, \dots, \rho_{eAttributes}|_{s_2}, \rho_{eClassifiers}|_{s_2}, \dots, \mathfrak{R}) \\
cl &:= (E_A|_{s_1 \between s_2}, E_{cl}|_{s_1 \between s_2}, E_C|_{s_1 \between s_2}, \dots, \rho_{eAttributes}|_{s_1 \between s_2}, \rho_{eClassifiers}|_{s_1 \between s_2}, \dots), \text{ with} \quad (3.54) \\
&\quad \forall(\varepsilon_i, \varepsilon), (\varepsilon, \varepsilon_j) \in \rho_e,\ \varepsilon_i \in E_{cl}|_{s_i},\ \varepsilon \in E_{te},\ \varepsilon_j \in E_{cl}|_{s_j} \quad (3.55) \\
&\quad\ \Rightarrow E_{\Gamma(\varepsilon_i)}|_{s_1 \between s_2} \cup \{\varepsilon_i\},\ E_{\Gamma(\varepsilon)}|_{s_1 \between s_2} \cup \{\varepsilon\},\ E_{\Gamma(\varepsilon_j)}|_{s_1 \between s_2} \cup \{\varepsilon_j\}, \\
&\quad\quad \rho_{\Gamma(\varepsilon_i, \varepsilon)}|_{s_1 \between s_2} \cup \{(\varepsilon_i, \varepsilon)\},\ \rho_{\Gamma(\varepsilon, \varepsilon_j)}|_{s_1 \between s_2} \cup \{(\varepsilon, \varepsilon_j)\} \quad i,j \in \{1, 2\},
\end{aligned}
$$

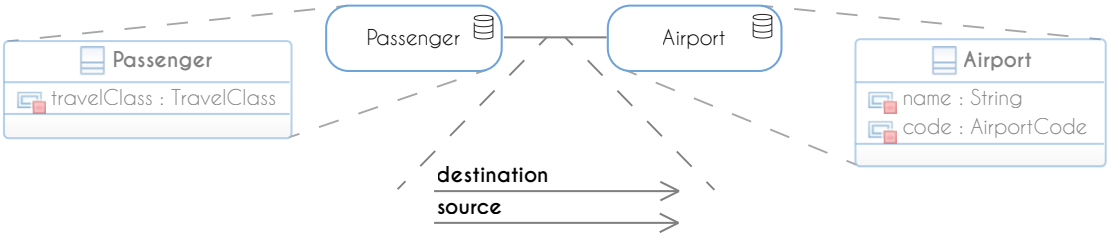


Figure 3.17.: Cross-links Example: Two adjacent `Models`

We use the "between" symbol ($\between$) for the restriction function (cf. equation (3.48)) to indicate that we refer to the elements between s_1 and s_2. These elements are `ETypedElements` in our models, according to figure 3.2 and table 3.1 on pages 34 and 35, which is a superconcept for attributes, parameters, and references. Hence, equation line 3.55 considers these typed elements (ε) as elements-in-the-middle, bridging two `Models`, and they belong in a cross-link if one of the associated elements is not in the same submodel. For our example, a cross-link $cl^{figure\ 3.17}$ could look like this:

$$
\begin{aligned}
cl^{figure\ 3.17} &:= (E_A|_{s_{Vehi} \between s_{Airp}}, E_{cl}|_{s_{Vehi} \between s_{Airp}}, \dots, \rho_{eAttributes}|_{s_{Vehi} \between s_{Airp}}, \dots) \quad (3.56) \\
E_C^{figure\ 3.17} &:= \{\varepsilon_C^{\texttt{Passenger}}, \varepsilon_C^{\texttt{Airport}}\} \\
E_R^{figure\ 3.17} &:= \{\varepsilon_R^{\texttt{Pass2Airp}}, \varepsilon_R^{\texttt{Pass2Airp}'}\} \\
\rho_{eReferences}^{figure\ 3.17} &:= \{(\varepsilon_C^{\texttt{Passenger}}, \varepsilon_R^{\texttt{Pass2Airp}}), (\varepsilon_R^{\texttt{Pass2Airp}}, \varepsilon_C^{\texttt{Airport}}), \\
&\qquad (\varepsilon_C^{\texttt{Passenger}}, \varepsilon_R^{\texttt{Pass2Airp}'}), (\varepsilon_R^{\texttt{Pass2Airp}'}, \varepsilon_C^{\texttt{Airport}})\}
\end{aligned}
$$

The reality suggests that `ETypedElement` is also the supertype to operations, so the simple `Connectors` we have provided are not enough. For example, picture a `Passenger` who embarks from a `Gate` to a `Vehicle`. This involves all three `Models` in our knowledge library in figure 3.4 (p. 48), and hence one `Connector` could not represent this information. Specifically, consider an additional method `embark()` for a `Passenger` with a `Gate` and a `Vehicle`. This parameter is shown in figure 3.18.

We can add the missing information by changing the cardinality of our `Connectors`.

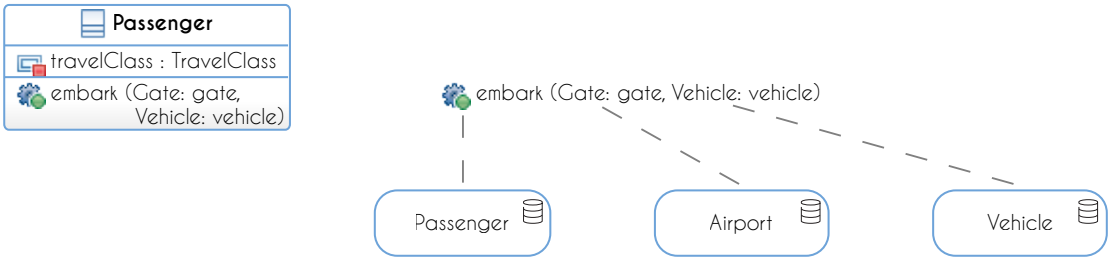


Figure 3.18.: Cross-links Example: Three adjacent `Models`

This is possible because they are represented as elements and not as relationships in our knowledge library (cf. figure 3.6 (p. 51)). Hence, separating our running example as above would no longer lead to an undetermined `Connector` for the `embark` method. Instead, this so called hyperedge allows a connection between multiple vertices [RWE13]. We can now relate all involved `Models` as shown in figure 3.19 and give a general definition of cross-links for a given model ($m := (E_A, E_{cl}, E_C, \dots, \rho_{eAttributes}, \rho_{eClassifiers}, \dots, \Re)$) and a set of submodels (s_i):

$$s_i := (E_A|_{s_i}, E_{cl}|_{s_i}, E_C|_{s_i}, \dots, \rho_{eAttributes}|_{s_i}, \rho_{eClassifiers}|_{s_i}, \dots, \Re), \quad i \in \{1, \dots\},$$

$$cl := (E_A|_{\natural s_i}, E_{cl}|_{\natural s_i}, E_C|_{\natural s_i}, \dots, \rho_{eAttributes}|_{\natural s_i}, \rho_{eClassifiers}|_{\natural s_i}, \dots), \text{ with} \tag{3.57}$$

$$\forall(\varepsilon_1, \varepsilon), (\varepsilon, \varepsilon_j) \in \rho_e,\ \varepsilon_1 \in E_{cl}|_{s_1},\ \varepsilon \in E_{te},\ \varepsilon_j \in E_{cl}|_{s_j} \tag{3.58}$$

$$\Rightarrow E_{\Gamma(\varepsilon_1)}|_{\natural s_i} \cup \{\varepsilon_1\},\ E_{\Gamma(\varepsilon)}|_{\natural s_i} \cup \{\varepsilon\},\ E_{\Gamma(\varepsilon_j)}|_{\natural s_i} \cup \{\varepsilon_j\},$$

$$\rho_{\Gamma(\varepsilon_1,\varepsilon)}|_{\natural s_i} \cup \{(\varepsilon_1, \varepsilon)\},\ \rho_{\Gamma(\varepsilon,\varepsilon_j)}|_{\natural s_i} \cup \{(\varepsilon, \varepsilon_j)\} \quad j \in \{2, \dots\},$$

$$\vee\ \forall(\varepsilon_1, \varepsilon), (\varepsilon, \varepsilon_x), (\varepsilon_x, \varepsilon_j) \in \rho_e,\ \varepsilon_1 \in E_{cl}|_{s_1},\ \varepsilon \in E_O,\ \varepsilon_x \in E_P,\ \varepsilon_j \in E_{cl}|_{s_j} \tag{3.59}$$

$$\Rightarrow E_{\Gamma(\varepsilon_1)}|_{\natural s_i} \cup \{\varepsilon_1\},\ E_{\Gamma(\varepsilon)}|_{\natural s_i} \cup \{\varepsilon\},\ E_{\Gamma(\varepsilon_x)}|_{\natural s_i} \cup \{\varepsilon_x\},\ E_{\Gamma(\varepsilon_j)}|_{\natural s_i} \cup \{\varepsilon_j\},$$

$$\rho_{\Gamma(\varepsilon_1,\varepsilon)}|_{\natural s_i} \cup \{(\varepsilon_1, \varepsilon)\},\ \rho_{\Gamma(\varepsilon,\varepsilon_x)}|_{\natural s_i} \cup \{(\varepsilon, \varepsilon_x)\},\ \rho_{\Gamma(\varepsilon_x,\varepsilon_j)}|_{\natural s_i} \cup \{(\varepsilon_x, \varepsilon_j)\}\ \ j \in \{1, \dots\}$$

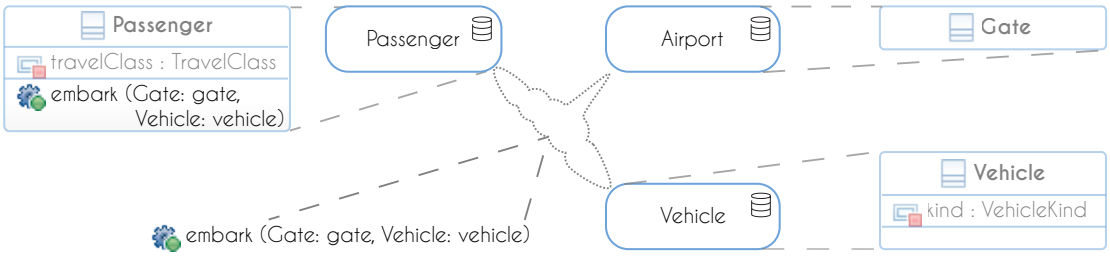


Figure 3.19.: Cross-links Example: “Hyperedge” with three adjacent `Models`

This time, we have a given set of submodels (s_i) and denote the restriction function as ($\natural s_i$), meaning that all elements are taken into account. Further, we fix the first submodel without loss of generality. We can do so because `ETypedElements` (E_{te}) can be treated as if they are contained by one `EClassifier` (E_{cl}). For example, a method

always belongs to a class, so we can take this submodel and denote it as s_1, as in line 3.58. All referenced `EClassifiers` (E_{cl}) now provide a bridge to related submodels, e.g., as an `EReference` (E_R). However, in our example, the additional condition is of interest, because it introduces operations. Here, ε is the operation and ε_x represents an `EParameter` (E_P); i.e., the `Gate` and `Vehicle` classes are parameters for the operation.

Finally, our cross-links are meant for more than pure relationships, and they should also allow for additional elements. For example, a cross-link could mediate between two `Models` as an adapter [Gam+95], which could be explained using the semantic information of a `Connector` (cf. figure 3.6 (p. 51)). Reiterating our example from figure 3.20, we find that a `Person` could hold a certain role in a `ClinicalRecord`, e.g., as a `Patient`. This adapter class provides additional information, e.g., a `PatientID`. This property is certainly specific to a `Patient`, but other classes could also adapt a person to a `ClinicalRecord`, e.g., a `Physician`. Exactly how a cross-link is defined is a design decision. The term `Physician` already implies that there are many kinds of them. Hence, it might be reasonable to store a hierarchy of `Physicians` in a `Model` instead. To the best of our understanding, there is no automatic way to determine classes that could play such a role, so we leave the editing of cross-links to the modeler.

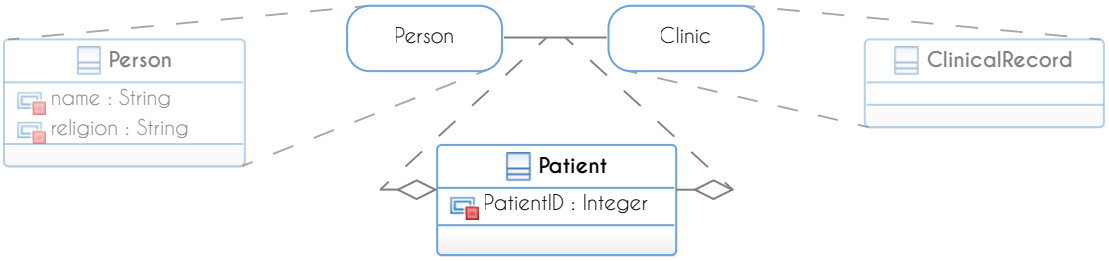


Figure 3.20.: Cross-links Example: Class between two `Models` (cf. figure 3.8 (p. 54))

Operation-Based Cross-links: An essential part of our approach is the interchangeable use of `Connectors`, either those that are persisted or in an operation-based manner. The latter is of particular importance for the reutilization of adjacent models. Here, the find operations (φ) prove beneficial, because they provide queried elements and forward them as parameters for other operations. Returning to the example discussed in figure 3.19, we can transform the `Connector` in an operation-based format ($\rightsquigarrow$) and apply it to the given or created `Models`. For instance, if we need the models first:

$$\Phi^{\text{AIRPORT}}_{\text{KL}\,\textit{figure 3.7}} = \varepsilon^{\texttt{Airport}}_{\text{M}} \rightsquigarrow \Im^{\texttt{Airport}}_{\textit{Ecore}}$$
$$\Phi^{\text{PASSENGER}}_{\text{KL}\,\textit{figure 3.7}} = \varepsilon^{\texttt{Passenger}}_{\text{M}} \rightsquigarrow \Im^{\texttt{Passenger}}_{\textit{Ecore}}$$
$$\Phi^{\text{VEHICLE}}_{\text{KL}\,\textit{figure 3.7}} = \varepsilon^{\texttt{Vehicle}}_{\text{M}} \rightsquigarrow \Im^{\texttt{Vehicle}}_{\textit{Ecore}}$$

Note that the knowledge library provides a `Model` element, and this contains the actual model that we implicitly transform into an operation-based form ($\Im$). A detailed operation sequence for our `Connector` applied on these models then takes place (cf. last line in

equation (3.60)). This sequence could proceed as follows (note equations (3.4) and (3.9) on page 38):

$$\begin{aligned}\mathfrak{I}^{\textit{figure 3.18}}_{\textit{Ecore}} =& \pi_{p^{set}}(\varphi^{\texttt{Passenger}}_{\mathsf{E_C}}, \{\varphi^{\texttt{embark}}_{\mathsf{E_O}}\}) \circ \\ & \pi_{p^{set}}(\varphi^{\texttt{embark}}_{\mathsf{E_O}}, \{\varphi^{\texttt{Gate}}_{\mathsf{E_P}}, \varphi^{\texttt{Vehicle}}_{\mathsf{E_P}}\}) \circ \pi_{c}(\mathsf{E}^{\texttt{embark}}_{\mathsf{O}}) \circ \\ & \pi_{c}(\mathsf{E_P}(\varphi^{\texttt{Gate}}_{\mathsf{E_C}})) \circ \pi_{c}(\mathsf{E_P}(\varphi^{\texttt{Vehicle}}_{\mathsf{E_C}})) \circ \\ & \mathfrak{I}^{\texttt{Passenger}}_{\textit{Ecore}} \circ \mathfrak{I}^{\texttt{Airport}}_{\textit{Ecore}} \circ \mathfrak{I}^{\texttt{Vehicle}}_{\textit{Ecore}} \end{aligned} \tag{3.60}$$

Equation (3.60) can be broken up into four steps. First, the last line of equation (3.60) uses results from our knowledge library ($\mathbb{KL}^{\textit{figure 3.7}}$) and is only provided to complete the example, but creates the models `Vehicle`, `Airport`, and `Passenger`. If these three models already existed, this line could be dropped. We will come back to this point later in subsection 3.5.5 (p. 127) and subsection 3.5.6 (p. 132), because only one of the three models may already exist. The other two will be created, which is sufficient because all our models in `Models` are complete in themselves and have no external references. The penultimate line creates the parameters. The second line of equation (3.60) creates our operation for the class and sets the parameters that have been created; finally, the operation is tight to the `Passenger` class. Note that we make excessive use of our short notation.

3.3.4. Saving Models

The final step in harvesting models deals with saving the separated submodels to a given destination handled by a saver (cf. figure 3.14). In our case, we consider a saver for a knowledge library that can handle `Models`, `Connectors`, cross-links, and `MetaInformation` (cf. figure 3.6 (p. 51)). Given `Models` with a user-input `name`, `description`, and `purpose`, each representing a respective submodel, we can employ our indexing (I) in equation (3.61) to place each submodel in our knowledge library ($\mathbb{KL}$) and add `Connectors` in equation (3.63). Further, we add a `Group` in equation (3.62). Note that information on `Categorys` can only be established if given by the user, so we omit this here. More formally, the sets and indexing look like:

$$\mathsf{E_M} = \mathsf{E_M} \cup \mathsf{E}^{\texttt{new}}_{\mathsf{M}},\ \mathsf{I}(\mathbb{KL}, \varepsilon^{\texttt{name-i}}_{le}),\ \varepsilon^{\texttt{name-i}}_{le} \in \mathsf{E}^{\texttt{new}}_{\mathsf{M}} \tag{3.61}$$

$$\mathsf{E_C} = \mathsf{E_C} \cup \varepsilon^{\texttt{new}}_{\mathsf{C}},\ \mathsf{I}(\mathbb{KL}, \varepsilon^{\texttt{new}}_{\mathsf{C}}) \tag{3.62}$$

$$\mathsf{E_C} = \mathsf{E_C} \cup \mathsf{E}^{\texttt{new}}_{\mathsf{C}} \tag{3.63}$$

Next to indexing, equations (3.61) to (3.63) produce no more than the unions of sets by postponing an explanation of the newly created sets. However, the individual elements of these sets need further explanation, as given in equation (3.64), for the `Models`:

$$\begin{aligned}\mathsf{E}^{\texttt{new}}_{\mathsf{M}} :=& \bigcup \varepsilon^{\texttt{name-i}}_{\mathsf{M}},\ \text{with } \varepsilon^{\texttt{name-i}}_{\mathsf{M}},\ \texttt{name-i} \in \textsc{str}, \\ & \varepsilon^{\texttt{name-i}}_{\mathsf{M}}.\texttt{files} := \mathsf{s}_i,\ \varepsilon^{\texttt{name-i}}_{\mathsf{M}}.\texttt{description},\ \varepsilon^{\texttt{name-i}}_{\mathsf{M}}.\text{“purpose”} \in \textsc{str}\end{aligned} \tag{3.64}$$

We have selected a middle ground between concept and realization here by assigning the submodel to the `files` attribute, whereas it should actually be a `URI`. Further, for simplicity, we have used a very basic notation for the `purpose`, because this information belongs to the `MetaInformation` that we will introduce in subsection 3.4.3 (p. 98). Additionally, we take elements for granted, i.e., given without instantiating them explicitly. We should introduce `Connector`s enhanced with cross-links if available:

$$\begin{aligned} \mathsf{E}_{\mathsf{C}}^{\mathtt{new}} = \bigcup \varepsilon_{\mathsf{C}}^{\mathtt{name\text{-}i}},\ &\text{with } \varepsilon_{\mathsf{C}}^{\mathtt{name\text{-}i}},\ \mathtt{name\text{-}i} \in \mathsf{STR}, \\ &\varepsilon_{\mathsf{C}}^{\mathtt{name\text{-}i}}.\text{source} := \varepsilon_{\mathsf{M}}^{i}, \varepsilon_{\mathsf{C}}^{\mathtt{name\text{-}i}}.\text{target} := \varepsilon_{\mathsf{M}}^{j}, \varepsilon_{\mathsf{M}}^{i}, \varepsilon_{\mathsf{M}}^{j} \in \mathsf{E}_{\mathsf{M}}^{\mathtt{new}}, 0 \leqslant i < j \leqslant |\mathsf{E}_{\mathsf{M}}^{\mathtt{new}}| \\ &\varepsilon_{\mathsf{C}}^{\mathtt{name\text{-}i}}.\text{syntactics} := \text{cl for } \varepsilon_{\mathsf{M}}^{i}.\text{files and } \varepsilon_{\mathsf{M}}^{j}.\text{files cf. equation (3.54) given m} \end{aligned} \tag{3.65}$$

Again, we have selected a middle ground for the cross-link assigned to the `syntactics` attribute, and leave it there instead of the `URI`. Note that a cross-link might be empty ($\varnothing$) if there are no elements between the submodels with respect to their model, as explained in the last line of equation (3.65). In subsection 3.5.5 (p. 127), this will become an important distinction for determining whether a cross-link is available. Other than that, we have not yet included the necessary relationships. However, we would like our knowledge library to maintain a graph structure and not contain isolated `Model`s. A semi-formal notation for building the relationships that define the new `Group` and interlink the submodels stored as `Model`s is as follows:

$$\begin{aligned} \rho_{\mathbf{e}\text{Group}} :=& \rho_{\mathbf{e}\text{Group}} \cup \bigcup (\varepsilon_{\mathsf{M}}^{i}, \varepsilon_{\mathsf{C}}^{\mathtt{new}}),\ \varepsilon_{\mathsf{M}}^{i} \in \mathsf{E}_{\mathsf{M}}^{\mathtt{new}} \\ \rho_{\mathbf{e}\text{Connector}} :=& \rho_{\mathbf{e}\text{Connector}} \cup \bigcup (\varepsilon_{\mathsf{M}}^{i}, \varepsilon_{\mathsf{C}}^{j}), \\ & \varepsilon_{\mathsf{M}}^{i} = \varepsilon_{\mathsf{C}}^{j}.\text{source} \vee \varepsilon_{\mathsf{M}}^{i} = \varepsilon_{\mathsf{C}}^{j}.\text{target},\ \varepsilon_{\mathsf{M}}^{i} \in \mathsf{E}_{\mathsf{M}}^{\mathtt{new}}, \varepsilon_{\mathsf{C}}^{j} \in \mathsf{E}_{\mathsf{C}}^{\mathtt{new}} \end{aligned}$$

Keen observers will have noticed how the equations above digressed with respect to notation. First, we cannot denote general terms that are provided by user input, e.g., names for `Model`s and `Group`s. Hence, expressions like `name-i` will have to do for now. Second, we changed the notation compared to operation-based models ($\mathfrak{Z}$) with respect to structure. Before, we were stricter with the graph structure inherited by models. For example, we created a class and an attribute and had to assign them via a relationship. Here, we consider this level of semi-formal notation sufficient, because the knowledge library is, contrary to models, eventually generated as code.

Our remarks above consider a given knowledge library, but our harvesting framework also allows other persistence destinations. In practice, a migration to a knowledge library might start from dedicated and indexed folders, go to shared folders on some server or version control repository, until a meta-structure like our knowledge library is introduced. A simple starting point would be a mega-model approach with querying functionality, and a final outcome could be a distributed knowledge library like ours with a dedicated graph server for structural information, an indexing server, and a version control server exposing the actual content, i.e., the models and meta-information.

3.3.5. Design Rationales and Observations

Our approach for harvesting models, which we introduced above, addresses the "Harvesting Challenge" described in section 1.2 (p. 6) and inherits some obvious drawbacks. These can be induced by the design decisions we outlined earlier, e.g., on our operation-based models or knowledge library, constrained by our project goals, e.g., the E, i.e., "easily" in HERMES, or others. We now discuss these, starting with the most pressing observations.

The most important observations concern scale and data format. First, and most pressing, is that this harvesting approach does not scale, but it is not meant to. We are comfortable with this, because models put away for reuse should not exceed a certain complexity, as mentioned in equation (3.52) in terms of c_{max}. This is intended to ensure the models can be understood more quickly, which, to our mind, eases reusability. Second, the format we use for submodels digresses from the perspective of operation-based models. This is for reasons of comprehensibility and because the tuple form is quickly transformable to an operation-based model. In fact, this will be necessary for applying a model after it has been picked for reutilization, as we show in subsection 3.5.8 (p. 142). Similarly, a transformation for cross-links is required, as we show in subsection 3.4.6 (p. 106), for aspects of evolution.

In discussing less pressing observations, we must keep in mind that our approach is meant to provide options to modelers and common grounds for extendability. In this respect, we locate our approach in software engineering with a focus on enabling modelers, and not in terms of providing a one-size-fits-all solution. The reason for this lies in the subjective manner of harvesting, whereby modelers need to make up their minds. We can, at best, support in one way or another by offering options and proposals.

With regard to the algorithms we have introduced, we mentioned agglomerative (i.e., bottom-up) and divisive (i.e., top-down) approaches without discussing details. This means we skipped any elaborate discussion on runtime analysis, semantics, or characteristics. Runtime is certainly of importance when harvesting from models with thousands of elements. Still, we consider this beyond the scope of this project, as it is a topic for software engineering or software construction, and harvesting only serves the purpose of completing the picture for this project (cf. section 6.2 (p. 189)). Nevertheless, the algorithms we have modified remain in the same class of complexity regarding big-O notation [Flo62; War62; KL70; NG04], and our own proposal only concerns elements once after creating a dependency graph.

Furthermore, our goal is for the modeler to have the last word, yet we could incorporate semantics, e.g., by means of scenario walks or model footprints [Sen+09; JGB11]. However, we consider these fields too large for a detailed discourse, and we generally have not involved related software in our prototype because of its unavailability or incompatibility, e.g., proprietary and closed-source software.

In addition, top-down or bottom-up algorithms lead to results with different characteristics depending on the evaluation criteria used to achieve "intra-cluster density" and "inter-cluster sparsity". Consider four graphs: a linked list, circle, binary tree, and a star

of n vertices. Now, combine them and place the circle and star at the end or in the middle of the linked list. Finally, consider a complete graph (K_n). Applying both kinds of algorithm provides a good sense of the strengths and weaknesses with respect to individual evaluation criteria. We omit this for the sake of brevity and because graph theory provides further details [AW12].

Additionally, hill-climbing algorithms, e.g., Kernighan–Lin clustering, might become trapped around local extrema, preventing the desired results, i.e., global extrema, from being attained. However, we did not include a randomizer, e.g., to achieve different balanced sets in the case of Kernighan–Lin clustering, as this is beyond the scope of the current project.

The important point about the algorithms introduced earlier is the way we treat models as dependency graphs and that this allows the application of well-known graph mining algorithms [AW12]. In addition, this enables other algorithms: without including further details, we could have delved into spectral clustering or quasi-clique clustering, but we prefer a simple approach that is functional without limiting its extensions. Further, we should keep in mind that we can use adjacency matrices interchangeably with the dependency graphs we have introduced.

Additionally, the introduced algorithms rely on certain cluster parameters. One example is the weights adjusted from Struber et al. [Str+14]. These are not our contributions and are left as parameters for adjustment. Our tests with the provided parameters demonstrate that they are reasonable choices, as also evaluated by the original authors. Note that other parameters, e.g., the threshold for submodel candidates (c_{max}), are generally meant to be set to counter information overflow and ensure that the eventual models are appealing [Stö14]. We will return to this point in subsection 3.4.4 (p. 101).

With regard to the way we model data, we have already distinguished our knowledge library with an ontology in subsection 3.2.4 (p. 65), but have only recently introduced some additional details. These details mostly comprise our cross-links, i.e., `Connectors` with syntactic information, how we build them, and what purpose they serve. In doing so, we underlined the importance of the information being preserved, because it is the `Connectors` that interlink with strong semantics. This is, to the best of our knowledge, not possible in other approaches, e.g., ontology. In addition, as soon as we introduce generations in subsection 3.4.6 (p. 106), we add more semantics to `Connectors` with syntactical information. Further, we will see how change propagation is handled. An alternative to our `Connectors` with cross-links could have been to drop this interlinking information during harvesting or have larger models in our knowledge library. This, of course, would make reutilization more tedious or less appealing [Stö14]. However, this option has been explored elsewhere, as we already elaborated when discussing the "Model Repository" in subsection 3.2.4 (p. 65) [Eli+10].

On a methodological note, which we largely omit (except in section 6.2 (p. 189)), some decisions leave the door open for migration to a knowledge library with cross-links serving operation-based model recommender systems. The starting point need be no more than a set of model files stored in a directory. Hence, our models stored in `Models` must be independent for the separation of concerns and model size.

Finally, we alluded to but did not elaborate on functions for determining the benefit or redundancy of models to a knowledge library. Combining both in a mechanism we could denote as “recommend-in” would be of interest, as we explain in our outlook (section 6.2 (p. 189)). For now, we exclude both because of their subjective nature given the approach explained so far. We provide the list of known elements ($\mathcal{K}_C$) and the modeler makes a conscious decision on whether to introduce redundancy or not. As a knowledge library grows in size, it is almost inevitable that the redundancy will also grow.

3.3.6. Related Work

Previous research approaches submodel extraction as a means of simplifying from a perspective of abstraction; this was summarized in the 1990s by Frantz [Fra95]. Further, the decomposition of ontologies was studied in the same decade by Wand and Weber [WW90].

Regarding the actual UML-model-slicing techniques, the roots are often attributed to program slicing by Weiser [Wei84], and a recent literature review on model slicing is provided by Blouin et al. [Blo+15]. For us, this paves the way for the embedding of other approaches as well as proposing our own solution. A closer look at model slicing approaches shows that several perspectives can be taken—some emerged from program slicing and others from transformations, grouping, graph traversal, crawling, matching, or clustering [Gör+13]. Strictly speaking, even the model footprints developed by Jeanneret, Glinz, and Baudry enable model slicing [JGB11].

Kagdi, Maletic, and Sutton were among the first to investigate UML-model slicing, and did so for large models from a maintenance perspective [KMS05]. They introduced the notion of context-free model slicing, because the class diagrams they investigated did not comprise behavioral information. The actual slicing is undertaken by an algorithm that terminates according to some given slicing criteria. These are formulated as a set of constraints, comprising an initial set, selected elements, and a dimension. The latter could be a maximum path length for a graph walk. Altogether, the slicing criteria control the given algorithm, and Kagdi, Maletic, and Sutton discuss a total of eleven properties regarding slices, e.g., a slice might contain gaps under certain conditions. Further, they discuss aspects of fault localization, metrics, and others. Many of their rationales guided our development, but the lack of interaction in their approach and their fundamentally different technological basis are not acceptable in our framework. They use an in-house open modeling framework tailored to UML built with Python.

Lano and Kolahdouz-Rahimi researched model slicing in terms of maintaining semantic equality while reducing complexity [LK10]. Hence, their results can be semantically isomorphic or structurally amorphous. They demonstrate their approach with UML-RSDS, which is a subset of UML with a focus on reactive systems that holds entrance entities called controllers. This provides a starting point for slicing, and additional constraints are given in OCL. They define data dependencies and write frames to ensure the validity of outcomes as required for UML. We return to the work of Ma, Kelsen, and Glodt [MKG15]. Their overall goal is model analysis, factoring, and enhanced comprehension. They go

into further detail regarding operation-based class diagrams, which are class diagrams aided by sequence diagrams, and state machines [LK11].

Another approach, which is built around UML models enhanced with OCL, is that presented by Sun, France, and Ray [SFR13]. They consider class diagrams in attempting to develop a quicker evaluation of OCL constraints in smaller models. To do so, they transform inputs in dependency graphs, derive slicing criteria, and extract fragments, which are submodels in terms of other related work and their figures. These fragments can contain redundancy, as they explain by an example modeling a location-aware role-based access control model. Altogether, they transform a class diagram with OCL constraints in a dependency graph, remove irrelevant elements, perform issue analysis, and decompose the dependency graph. These aspects proved valuable for our design, but the extent of redundancy they considered exceeds our requirements, which is not surprising given their desire to speed up OCL evaluation.

An approach that combines available UML models of all kinds into a Model Dependency Graph (MDG) has been presented by Lallchandani and Mall [LM08]; an extended version of their contributions introduces Class Dependency Graphs (CDGs) [LM09]. MDG "integrates various UML diagrams into a single system model" [LM08], which is presented by class diagrams and sequence diagrams. The graph structure is quite congruent to the results of CMOF and enables architectural model slices to be computed by means of an algorithm. Their algorithm basically traverses the MDG and produces static or dynamic slices. The latter is an extract of the MDG produced by means of a scenario. CDG focuses on transformed class diagrams with the exception of associations, because Lallchandani and Mall claim that those are represented by calls modeled in sequence diagrams that impact on Sequence Dependency Graphs (SDG). Further, they present a Static Slicer for UML Architectural Models (SSUAM) that implements their approach and provide measures regarding the runtime overhead produced by transforming models into CDGs/SDGs [LM09]. Compared to our approach, they provide automated model slicing with few means of interaction. This contradicts our assumption that semantic aspects are vital for submodel creation.

Sen et al. ease the "meta-muddle" by pruning, whereby they restrict a model and its respective metamodel to the bare minimum [Sen+09]. This is motivated by scenarios they provide that mostly focus on transformation. The limited models and metamodels they are looking for are called "effective metamodels", and are gained by their proposed pruning algorithm. Working with MOF and EMF/Ecore, their implementation is based on Kermeta, a modeling framework for aspect-oriented modeling [MFJ05]. Kermeta and transformations explain why our approaches are incompatible.

Similarly, Bae, Lee, and Chae proposed a slicing approach for the UML metamodel with respect to diagram types [BLC08]. They propose to transform the UML model into a directed multi-graph that can be modularized for, e.g., use-case diagrams and class diagrams. This fosters comprehensibility and tool development through simpler models. To that end, they work solely on generic information and not the actual models that we consider. This means they provide ready-to-use sets, which is not an option in our case.

In terms of scalability by partitioning leading to formal verification checks, Shaikh et al.

investigated model slicing given a property [Sha+10]. In a sense, they take a model and OCL constraints, slice them, check them more or less independently, and combine the verification results to give an overall result. This explains why different approaches are required to solve this verification task, although, similar to our approach, submodels are built. The slices that are built hold logical properties, which are different to our syntactical and structural properties.

Struber et al. incrementally split a model into submodels, concentrating more on semantics than graph structure [Str+13; Str+14]. In their example, they give a model and leverage textual descriptions to enable semantic splits through an information retrieval mechanism. The results are enhanced by model crawling with experimental weights for relationships, so users can adjust the proposed set of submodels. We use their weights for traversal, but have no textual descriptions, so their information retrieval approach cannot be applied in our case. Otherwise, the steps and formalities lead in a similar direction, though our purpose and operation-based view builds a different foundation. Further, we offer cross-links and do not harvest entire models. We will return to term frequency and inverse-document frequency when describing reuse.

Becker and Gruhn envision a grouping mechanism for models, so reuse is simplified and encouraged [BG10]. They adapt approaches from natural language processing and classify the identification of model characteristics in text, structure, semantics, and automatic feature deduction. Their environment is the SAP R/3 reference model, so further publications are sparse and follow-ups appear to have changed to component-based service modeling.

Kompren is a DSML that enables model slicing by means of Kermeta for arbitrary metamodels and was constructed by Blouin et al. [Blo+11]. To that end, it is a generic approach, including automatic generation of model slicers, and is meant for three use cases: first, determining the effective metamodel of a model; second, semantic zoom with regard to a given aspect; and third, runtime model monitoring to highlight currently used elements. Therefore, slicing considers metamodels and their instances by offering two different modes for slicing: strict and soft. The former retains all structural constraints, whereas the latter allows some to be weakened. Further, a radius can determine the distance from a selected class that is to be considered. Blouin et al. extended and refined their work in several respects, e.g., properties of generated slicers and related work [Blo+15]. Our approach does not deal with different levels of models, and the metamodels in their method are dealt with as models in our approach.

On a more formal level, there are approaches similar to our enhanced knowledge graph. For example, Konrad Voigt follows the idea of separating large models for the purpose of model matching [Kon11]. Hence, clustering approaches employing divide-and-conquer techniques for graphs are implemented. For example, Planar Edge Separator (PES) is adjusted to solve metamodel matching by means of relationship weights. Compared to our approach, Konrad Voigt focuses on automation and avoids adjustments for submodels, i.e., skips the candidate adjustment step.

Kelsen, Ma, and Glodt provide a formal foundation for metamodels, models, and submodels, as we do, but simplify submodels to be subsets of models adhering to

constraints [KMG11]. These are the type compatibility regarding metamodel constraints, called forward constraints, and multiplicity with respect to source and target cardinality. They prove the correctness and soundness of their approach by means of fragmentable links, which establish correspondences between types in models and their use, similar to parameters, attributes, or associations. The same group generalized and extended this work to be more generic and aimed for "model management by decomposing complex models into smaller submodels" [MKG15]. Their realization is bound to EMF, so they include formalities tailored for Ecore, called EMF metamodels, for models and submodels. Once more, they prove the soundness and correctness. It is useful that EMOF and Ecore differ in the respect discussed in subsection 3.1.1 (p. 33), because this renders all attributes and references fragmentable.

3.3.7. Summary of Harvesting Models

Harvesting models, as introduced in this section, comprises four steps. First, finding known parts; second, building submodel candidates; third, user alteration; and fourth, separating and storing submodels as `Models`. Therefore, we introduced the set of *known elements* ($\mathcal{K}$) to work with, and semi-formally defined the term submodel (s) and a relationship (s $\subseteq$m). We then introduced dependency graphs (DG) as the foundation for submodel candidate generation, i.e., graph clustering. We offered several simple algorithms for building submodel candidates, some using common algorithms from graph or network theory, and discussed the common problem behind this, including related parameters. We deliberately kept this discussion simple, because semantics is important for the final submodels and user opinion seems more important than digging into more sophisticated approaches, e.g., applying spectral clustering, quasi-clique clustering, or others [AW12]. After that, we introduced cross-links (cl) as a restriction function ($\bowtie s_i$) and provided an operation-based view on cross-links. Aspects that have not been discussed, though they are important for a realization, include stop words and stemming [Por80; Wil06; MRS08]. These techniques allow us to reduce redundancy and simplify the indexing, as mentioned in subsection 3.2.2.

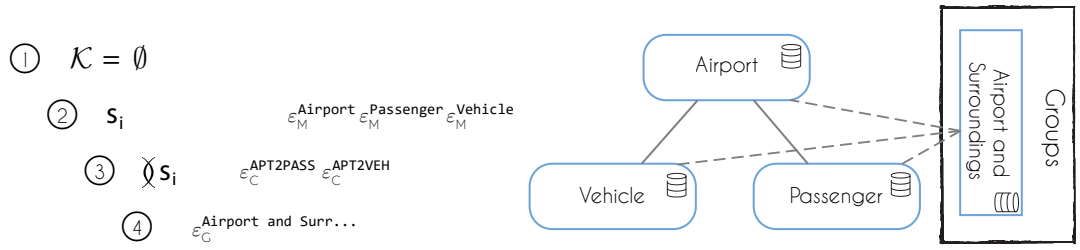


Figure 3.21.: Harvesting Example Summarizing equations (3.61) to (3.63) for figure 2.3

3.4. Evolving Models

According to the *Encyclopaedia Britannica*, evolution is a "theory in biology postulating that the various types of plants, animals, and other living things on Earth have their origin in other preexisting types and that the distinguishable differences are due to modifications in successive generations" [Saf14]. However, biology is not the only science researching evolution, and chemistry, linguistics, and others postulate their very own theories for various aspects under investigation.

For evolution in computer science [MFP06], artifacts "have their origin in other preexisting types" and similarly expose "distinguishable differences [] due to modifications" [Saf14]. Here, two terms emerged as virtual synonyms in the 1960s [GG08]: maintenance and evolution. In the 1980s, the former was grouped into perfective, corrective, and adaptive changes [LS80]. Eventually, this led to the ISO/IEC standard 14764:2006 adopting these classifications [III06]. The latter, evolution, was researched at the same time, leading to "Laws of Software Evolution" for different types of software [Leh80].

Maintenance and evolution are still often used synonymously, but experts agree there is at least a fine line of distinction [BR00; Cha+01; GG08]. Whereas maintenance is considered to cover activities in deployment scenarios, evolution, initially thought of as unexpected [BL76], includes the construction of new and enhanced versions, i.e., development that replaces artifacts in maintenance. A more recent understanding of maintenance and evolution builds on the "staged model" of software lifespan [GG08; Raj14], which puts evolution chronologically before maintenance [BR00]. In other words, maintenance makes no major enhancements, but only small fixes to prevent phase-out.

On a more specific note for software artifacts, evolution is studied in terms of models and metamodels [WK08; Kög11; Lev+11]. This introduces aspects of co-evolution for models, as respective metamodels change [Cic+08; Her11]. Fortunately, this is not an issue for our knowledge library, but the idea is inspiring, as we show later.

For now, we note that evolution is no longer regarded as unexpected, but is often seen as undirected, being determined by unanticipated new requirements [BR00; Cha+01; GG08]. For a knowledge library, this is a hindrance, because requirements, once set, must persist. This allows a change of focus to quality and guidance for model evolution. If a model is placed in our knowledge library as a `Model`, then editing sequences (σ) change and hopefully improve its quality and overall correctness. This can go on for some time, continuously changing the quality for better or worse, which we summarize in a status ($-$,$\circ$,$+$). A `Model`, as timed snapshots evolving in our knowledge library, could roughly be sketched as follows:

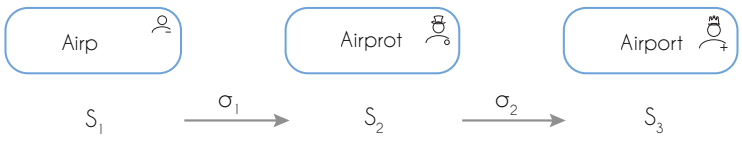


Figure 3.22.: Evolution Sequence Example for `Airport` from figure 2.3 (p. 27)

Although the sketch above omits an actual representation of a model, one is indicated with no more than a box. Further, the only indicator for change is a reedited name. Still, this should convey the general idea we lay out below, where we adapt and expand our previous work to the framework we have presented so far [Rot+13; Gan+13; Gan+16].

3.4.1. The Model Evolution Framework

On a conceptual level, our evolution approach comprises a central component, which we call the *model evolution framework*. This is surrounded by four supporting parts, as depicted in figure 3.23. The *versioning* we employ is shared with the data framework introduced in subsection 3.2.1, but is enhanced with additional semantics, as we show later. The *reviews* and *metrics* are supporting mechanisms in regard of quality assessment, which is required for guidance, as introduced in subsection 3.4.5. Finally, a *UI* exposes the concepts developed and provides user assistance, as we show in section 4.4 (p. 155).

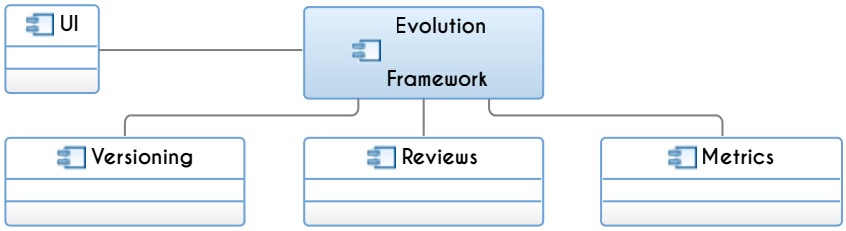


Figure 3.23.: Model Evolution Framework (MEF)

Subsequently, we go into further detail regarding the evolution framework from figure 3.23 and explain the concepts it comprises. Therefore, we establish an understanding for evolution in knowledge libraries, add quality aspects, and use them to introduce proactive quality guidance for knowledge libraries by extending the concepts from figure 3.6 (p. 51) with those depicted in figure 3.30. We illustrate this with an excerpt from our running example in figure 2.3 (p. 27) as an evolved sequence in figure 3.24.

3.4.2. Model Evolution in Knowledge Libraries

We started this section with a definition of evolution that emphasized “modifications in successive generations” [Saf14]. This is true for biology, but was put differently by Lehman, who describes evolution for software artifacts as a “process of changes” [Leh80]. For models, this was narrowed to the change primitives “add”, “delete”, “rename”, and “retype” by Keienburg and Rausch [KR01]. We can map these to the operations introduced in subsection 3.1.2 (p. 37), i.e., π_c, π_d, $\pi_{p^{set}}$, and $\pi_{r^{ass}}$. For the sake of convenience, we define the scope of this section as $\pi_{add} := \pi_c$, $\pi_{del} := \pi_d$, $\pi_{ren} := \pi_{p^{set}}$, and $\pi_{ret} := \pi_{r^{ass}}$. However, Keienburg and Rausch pursue and regard a new model version, as the result of “proceeding [an] ordered list of model change primitives on an existing [] model” [KR01]. For us, this list is equivalent to a sequence of operations (σ) and the version is a model

snapshot (S). Both are illustrated as an overview in figure 3.22 and detailed in figure 3.24. In other words, model snapshots represent different versions of a model, and figure 3.22 or figure 3.24 depict three model snapshots denoted S_1, S_2, and S_3. Note that the indexes are formally numerical version identifiers in ascending order, but can be replaced with something semantically stronger, as we show later.

Providing more semantics for the sequences of operations, we introduce the term model evolution steps. These are the transitions between snapshots representing the operations applied to a model snapshot to become a successive snapshot. Figure 3.24 shows two evolution steps, denoted σ_1 and σ_2. Again, the indexing with numbers is ascending, although semantic values could be used, as introduced for sequences of operations in equation (3.18) (p. 40) for operation-based models ($\Im$).

Together, an alternating sequence of arbitrarily many model snapshots and evolution steps (S_i, σ_i, S_j, σ_j, S_k, σ_k, ...) forms model evolution. A model snapshot with no successive operation sequence (S_1, $\varnothing$) represents a lack of evolution as much as an empty model snapshot does ($\varnothing$, $\varnothing$). For figure 3.24, this means that model snapshot S_1 has evolved by means of π_{add} and π_{ret} operations to snapshot S_2, and, eventually, to S_3.

Careful readers might have noticed that the model snapshots in figure 3.24 look like vertices and the model evolution steps are represented as edges, just as in a graph. We did so to introduce model evolution represented as model evolution graphs $(V, E, snap, step)$, where V is a set of vertices, E is a set of edges, and *snap* and *step* are labeling functions. The first, $snap : V \to \text{IDX}$, labels each vertex to some co-domain $\text{IDX} \cong \mathbb{N}$, i.e., is countable and ordered. The second, $step : E \to \bigcirc_{i=0}^{n \in \mathbb{N}} \pi_i(\cdot)$ (cf. equation (3.15) (p. 39) given an undetermined operation-based model), labels each edge with a sequence of operations, i.e., forming an evolution step.

Example: For our example from figure 3.24, omitting the required model parameter for the labeling functions, we obtain the following:

$$
\begin{aligned}
V &:= \{S_1, S_2, S_3\} \\
E &:= \{(S_1, S_2), (S_2, S_3)\} \\
snap &:= \{(S_1 \mapsto 1), (S_2 \mapsto 2), (S_3 \mapsto 3)\} \\
step &:= \{(\sigma_1 \mapsto \pi_{ret}(\varphi_{E_A}^{\texttt{Airport.}name}, \{\varphi_{E_D}^{E_{String}}\}) \circ \pi_{add}(E_C^{\texttt{Runway}}) \circ \ldots \circ \pi_{add}(E_C^{\texttt{Tower}}), \\
&\quad (\sigma_2 \mapsto \pi_{del}(\varphi_{E_C}^{\texttt{Person}}) \circ \ldots \circ \pi_{ret}(\varphi_{E_R}^{\texttt{Terminal2Checkin}}, \{\varphi_{E_C}^{\texttt{Checkin}}\}) \circ \\
&\quad \pi_{p^{set}}(\varphi_{E_C}^{\texttt{Terminal}}, \{\varphi_{E_R}^{\texttt{Terminal2Checkin}}\}) \circ \pi_{add}(E_R^{\texttt{Terminal2Checkin}})\}
\end{aligned}
\tag{3.66}
$$

Note that operation sequences for the *step* function read from right to left. Further, we have omitted some obvious operations if a similar one is shown. For example, in equation (3.66) for σ_1, only two of the four elements are added, while for σ_2, only one out of three new relationships is added, which already requires three operations. Finally, the labeling function *snap* is not surprising, as it simply unveils the index of a snapshot identifier, but this is just a matter of notation.

Discussion: At the start of this section, we learned that evolution can be easily confused with maintenance, but model evolution, as introduced above, differs from model

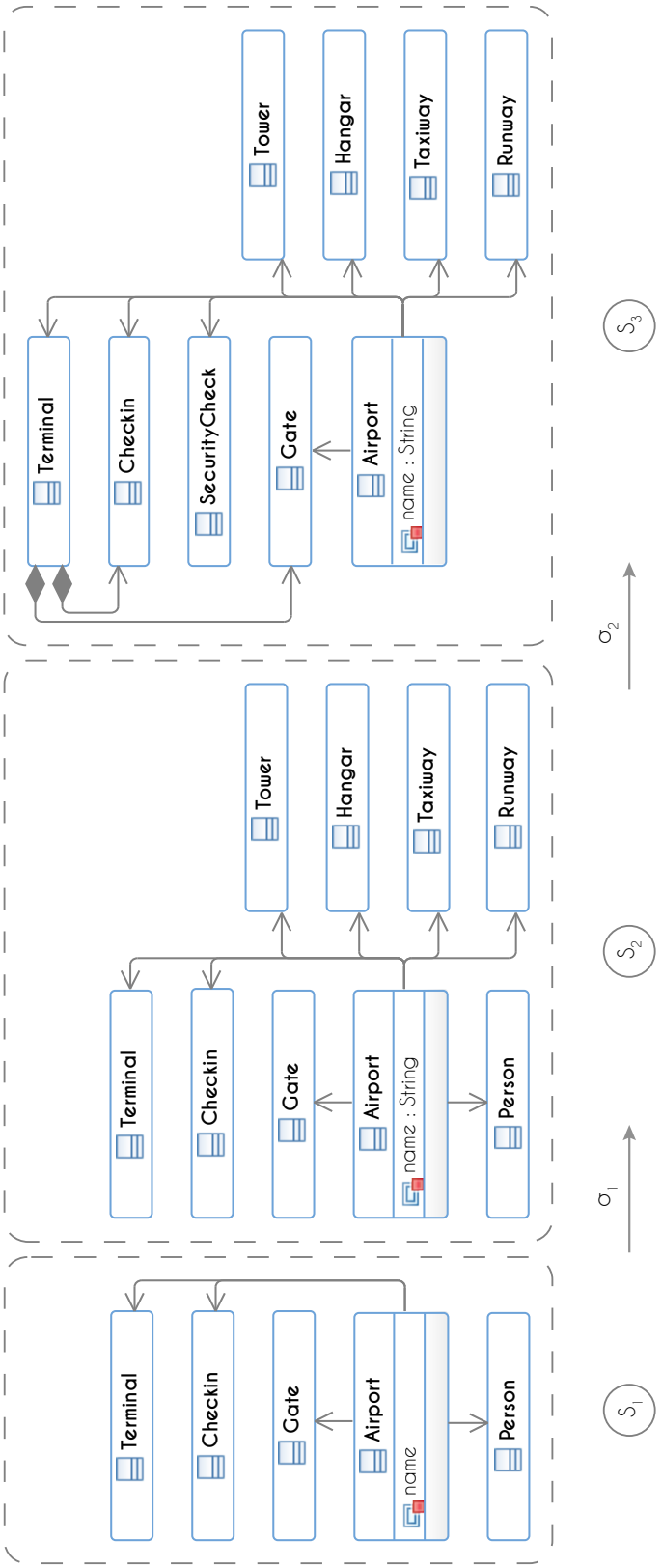


Figure 3.24.: `Airport` Snapshots, [Gan+16, similar]

maintenance in several respects. First, the latter succeeds evolution, summarizing only small fixes without major enhancements, because maintenance ought to preserve interface integrity as much as possible [JFC04; Bri+06; Eic+01; MFP06]. However, for a knowledge library, this is not an issue. Changes might and must be radical if the reusability of elements requires improvement. Second, maintenance leads, eventually, to phase-out and retirement, whereas evolution always allows new versions. Still, models represented as `Models` in a knowledge library might deprecate by virtue of not being used anymore. Moreover, we learned that the reasons for evolution are threefold [JFC04]. First, it might be to adapt to changed requirements. Second, it might be to correct or fix issues. Third, it might be to perfect the design or quality. We will come back to each of these reasons later, because the model evolution graph introduced so far is almost equivalent to version control graphs, but does not support quality assurance functionality. For that, we will traverse the evolution graphs and label vertices with both numbers and quality statements.

3.4.3. Model Evolution Stages

Biology distinguishes evolution between species and their branches. For example, the timeline of human evolution comprises several such branches until homo sapiens emerged. Though not always perfectly distinguishable, branches provide points in time for successive generations. For models, we introduce a similar mechanism to provide a statement about the quality of a model represented by a `Model`. In a nutshell, we will take a model evolution graph and add a quality statement to each and every model snapshot given the associated `Model` it represents.

The main quality concern for our knowledge library is reusability, so we have developed an assessment representing three respective degrees ($-$,$\circ$,$+$) [Gan+16]. We used this notation in figure 3.22, with "$-$", "$\circ$", and "$+$" denoting low, medium, and high reusability. As an additional representation, we chose the traffic light metaphor to keep the cognitive load low. This is because the number is small and traffic lights are well known and widely accepted. Thus, "$-$" can be represented by red, "$\circ$" by amber, and "$+$" by green. However, for a good representation in terms of Moody's "'Physics' of Notation", we added a textual representation to each [Moo09]. Hence, we consider the reusability denoted by "$-$" as `vague` or sketchy, "$\circ$" as `decent` or provisional, and "$+$" as `fine` or stable.

We refer to model reusability that is `vague`, `decent`, or `fine` as the model reusability stage, or simply stage, of a `Model`. In more detail, a `Model` in the `vague` stage is undetermined regarding its reusability and is, therefore, labeled red. This is the case for freshly added `Models`, because they have not yet been assessed. Often, this means that some processing is required to make the `Model` easier to reuse. For example, a model that has been freshly added to a knowledge library as a `Model` might be sketchy and expose unnecessary prefixes or suffixes, technology-induced elements, e.g., DAOs, or errors might need fixing. Hence, a `Model` in the `vague` stage is often the starting point for evolution and might be reused, but only with extra caution. Next, a `Model` in the `decent` stage represents a `Model` in which the major issues have been fixed, and is

considered generally reusable in most cases. Some issues may not have been dealt with, hence the amber label indicating that it is provisional and caution is necessary. For example, the `purpose`, which we see as a lightweight specification, might not be perfectly in line with the actual requirements of this respective model, the layout might be chaotic, or some design decisions could be improved. Altogether, a `Model` in the `decent` stage has reasonable reusability. Finally, a `Model` in the `fine` stage is labeled green for a good reason—it is considered stable and "almost perfectly reusable". Now, the `purpose` matches the intention of the respective model and all issues are resolved. Still, this does not guarantee that this model needs no alteration for reuse. It may require a design pattern [Gam+95], or could come with template information that needs to be filled in (cf. figure 3.6 (p. 51)).

Note that our example in figure 3.22 simplifies model reusability stages and shows a different stage per snapshot. In reality, sequences of snapshots with the same model stage might occur. In fact, this is likely given the quality assurance mechanism we introduce later, because reviews might urge a `Model` to remain in a particular stage until certain issues are resolved.

Sometimes, however, issues might not be resolvable or a model may no longer be intended for reuse if an alternative has emerged. In such cases, the `Model` remains in its reusability stage, but gets an extra flag signifying it has been deprecated. The semantics is similar to programming libraries, and denotes that the item is either already or soon to be replaced. Release notes often state this, and we introduce our mechanism later. Other than that, unresolvable issues, as mentioned above, might serve as counterexamples.

Evolution Stage Automaton: The model reusability stages introduced above inherit dependencies. Some of them can be derived from the explanations above, and the intended order between `vague`, `decent`, and `fine` is already implied by the meaning of low, medium, and high reusability or, in other words, by the connotations of sketchy, provisional, and stable. Thus, a linear order is apparent. Further, we have already mentioned that open tasks and issues might keep a `Model` in one stage, but what other transitions can be drawn between the reusability stages? We now describe the relationships depicted in figure 3.25.

Relationships are divided into those that are allowed and those that are prohibited. First, the semantics of stages implies that some decay can occur. This is possible in the opposite direction, as explained above, as well as from a `fine` stage to `vague`. For example, a `Model` in the `fine` stage may have grown to the extent that some parts have been extracted to another `Model`. This must leave both the remainder and the new `Model` in the `vague` stage. Second, one transition between stages must be prohibited, namely from the `vague` to the `fine` stage. We will go into further detail about the reasoning later, but we can now state that a model needs to undergo some quality assurance mechanism in terms of `purpose` and design rationale checks to become highly reusable, i.e., a `Model` in the `fine` stage.

Altogether, we have the evolution stage automaton depicted in figure 3.25. Formally, this comprises $A_{staged} = (Q, \Sigma, Z, \delta, q_0, F)$, where $Q := \{\texttt{vague}, \texttt{decent}, \texttt{fine}\}$ is the set of vertices, $\Sigma := Q \times (S \cup \{id_M\})$, with $S := \{\pi_{add}, \pi_{del}, \pi_{ren}, \pi_{ret}\}$ is the set of input

symbols, q_0 ($q_0 := \text{VAGUE}$) is the starting state, and F ($F := \varnothing$) is the empty set, as we do not consider evolution terminating. The set of transitions (δ) comprises $\delta := \delta_v \cup \delta_d \cup \delta_f$, with sets of transitions for each stage:

$$\delta_v := \{(\texttt{vague}, \pi_i) \mapsto \texttt{vague} \mid \pi_i \in S\} \cup \{(\texttt{vague}, id_M) \mapsto \texttt{decent}\}, \tag{3.67}$$

$$\delta_d := \{(\texttt{decent}, \pi_j) \mapsto \texttt{decent} \mid \pi_j \in S\} \cup \{(\texttt{decent}, id_M) \mapsto \texttt{vague}\} \cup \{(\texttt{decent}, id_M) \mapsto \text{FINE}\}, \text{ and} \tag{3.68}$$

$$\delta_f := \{(\texttt{fine}, \pi_k) \mapsto \texttt{fine} \mid \pi_k \in \{\pi_{ren}, \pi_{ret}\}\} \cup \{(\texttt{fine}, id_M) \mapsto \texttt{decent}\} \cup \{(\texttt{fine}, id_M) \mapsto \texttt{vague}\} \tag{3.69}$$

Any model added to our knowledge library as a `Model` starts in the `vague` stage (q_0), as mentioned above. Then, as the model is edited, i.e., operations (π) are applied, the `Model` remains in this stage until an identity operation (id_M) transfers it to the next stage. Alternatively, an identity operation can keep a `Model` in its current stage. We build in this non-determinism as a technicality. It serves two purposes: first, it is a means to create a snapshot if a `Model` remains in an stage, and second, it serves as a basis for the quality gates we introduce later. Quality is also the reason why a `Model` that has progressed to the `fine` stage can only remain in this stage if rename and retype operations are applied. The other operations, i.e., add and delete, place it in the `vague` or `decent` stage, because they can alter the respective model substantially, as we explain later.

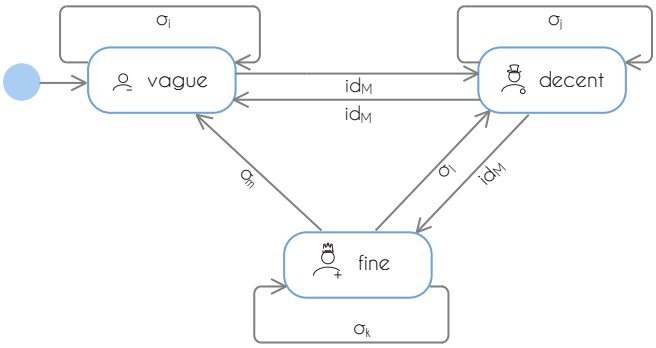


Figure 3.25.: Evolution Stage Automaton similar to [Gan+16]

Example: We can look at our model evolution graphs, or rather sequence, from figure 3.24 with the overview given in figure 3.22 and picture it as a settlement of an evolution stage automaton. The `Model` starts in a `vague` stage when it is added to the knowledge library and operations (σ_1) are applied (cf. equation (3.66)). The `Model` then progresses by means of an identity operation to the `decent` stage and so on. Note how the indicators ($-$,$\circ$,$+$) and symbols between figures 3.22 and 3.25 match.

More specifically, operations from σ_1 (cf. equation (3.66)) add classes and fix an issue with the `name` attribute by correcting the type to `String`. With the `Model` in the `decent` stage, operations from σ_2 add more classes and correct some semantic issues by removing the `Person` class, because this does not contribute to the model's `purpose`.

3.4.4. Quality for Model Evolution Stages

The measurement of software artifacts has a long history with a wide tradition of discussion and disagreement [Arb11]. For conceptual models, this is attributed to evaluation that "is by nature a 'social' rather than a technical process [and] inherently subjective" [Moo05]. In addition, quality characteristics and measurements are "designed for particular purposes" in general and "need to be calibrated" [Arb11].

Given a knowledge library with evolving models for reuse, we have a restricting environment for quality and measurement with specific requirements. This allows the development of a quality model with reuse as a primary quality goal and the tailoring of quality characteristics. As an example, we can set bounds on the size of models, because those too complex become unappealing for reuse [Stö14].

To design our quality model, we use a common framework comprising three quality dimensions, namely syntactic, semantic, and pragmatic quality [LSS94]. In addition to the common framework, we add emotional quality to our quality dimensions. Each quality dimension can be decomposed as depicted in figure 3.26.

Syntactic quality refers to errors in the sense of misspelling. This can either be a typographic error or a malformed keyword. While the former requires a spell checker for a given language, which might not be sufficient, the latter can be automatically processed in the following respects, i.e., quality characteristics. First, metamodel conformity states whether a model is valid in the sense that it uses words of the given metamodel to form a model. For EMF models, this means that they conform to Ecore. Second, transformability requires a generator that can transform a model into another representation, e.g., source code, or our operations in a graphical sense [DGL13]. This is important because certain rules must be followed, but these are sometimes not expressed in the respective grammar. In EMF, some constructs of collections are better expressed as lists instead of arrays. Finally, defect-freeness requires a well-formed model in the sense that the syntax is correct. For XMI models, this means that all opening elements have corresponding closing elements, e.g., each "$<$" is eventually followed by a respective "$>$".

Semantic quality comprises quality characteristics that try to grasp whether a model conveys the content it should, for example, whether the statements of a model correspond to its domain. First, semantic validity means that all statements of a model are valid with respect to the given domain, and invalidity occurs for statements where their removal is more beneficial than their retention. In a sense, this means that deletion must be better. Second, completeness means that, in contrast, for any potential statement, the disadvantages of not including it are outweighed by the advantages of doing so. In a sense, this means inclusion must be better. Third, confinement states that a model comprises enough valid statements to convey an intention or solution for a given problem. In a sense, this means it works as an idea.

Pragmatic quality relates to the comprehension of model content. First, understandability is the degree to which a modeler's interpretation is congruent to the model's intention. Second, maintainability denotes the degree to which a model can be understood so that it can be reused in a new environment. Third, purpose extraction reflects whether the model's intended `purpose` corresponds to the formulated `purpose`, our lightweight

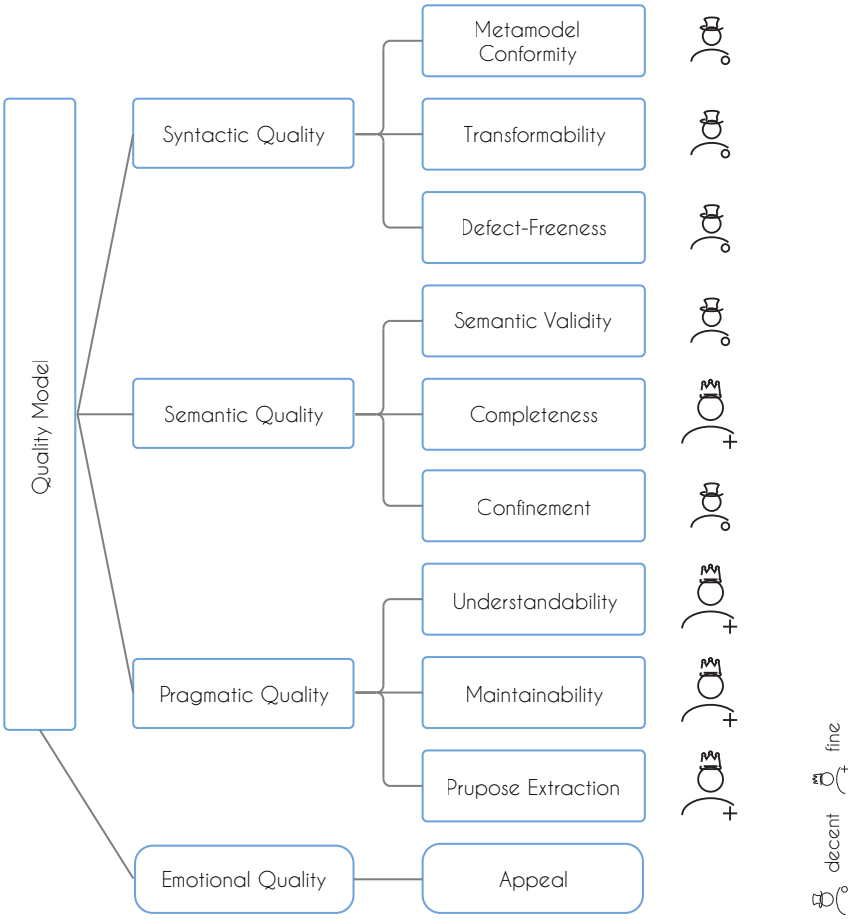


Figure 3.26.: Quality Model similar to [Gan+16]

specification. Emotional quality is, admittedly, a fuzzy quality dimension that grasps the attitudes towards a model. We believe that the appearance of models is an indicator for that, because appealing models are more likely to be reused. Hence, we count the actual reuses and “Likes” we gather from the UI. We distinguish the representation of emotional quality in figure 3.26, because we are aware of its subjective nature.

Quality Gates: The quality model presented above serves as a first enabler and eliminates some of the non-determinism introduced in the evolution stage automaton as a technicality. We do so by introducing checkpoints that require certain quality characteristics to be met in order to be passed. Figure 3.27 provides an idea of how we enrich our evolution stage automaton from figure 3.25 with such checkpoints.

These checkpoints are called quality gates (QG) in our approach, and are meant to structure and guide model evolution in knowledge libraries. This is possible by defining a quality gate to be a set of quality characteristics from our quality model that must be satisfied to be passed. Figure 3.26 depicts our symbols for reusability stages next to each quality characteristic, which means that this quality characteristic is a requirement

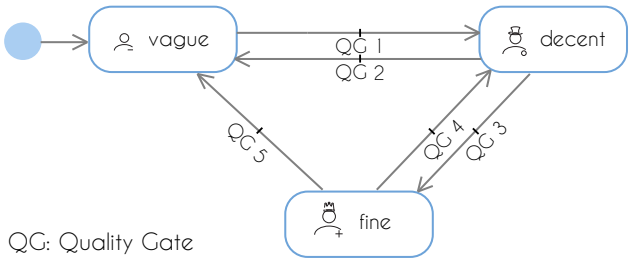


Figure 3.27.: Quality Gates similar to [Gan+16]

before a model can reach either the `decent` or `fine` stage. Of course, all characteristics required for a model to obtain the `decent` stage are also required to reach the `fine` stage.

From figure 3.25, we know that every model represented by a `Model` starts its evolution in the `vague` stage with, hopefully, the goal of reaching the `fine` stage. Though the former does not require any quality characteristics to be met, the latter requires all of them. Thus, the distinguishing quality gates, according to figure 3.27, are QG 1 and QG 3. The former requires metamodel conformity, transformability, defect-freeness, semantic validity, and confinement. In other words, a model represented by a `Model` passing QG 1 is valid, can be an input for a code-generator, is well-formed, does not contain too many domain statements, and is sufficient to provide a decent or provisional solution (amber). The latter requires a `Model` to be in the `decent` stage, i.e., meet all requirements for QG 1, and also requires completeness, understandability, maintainability, and `purpose` extraction to hold. Put differently, a `Model` passing QG 3 comprises everything necessary to solve the intended problem, conveys what it intends to solve, is adaptable to intended environments, and has a `purpose` describing exactly the model's intention. As a result, it can be considered a fine or stable solution (green).

However, `Models` need not progress in a linear direction as described. In case a `Model` is in the `decent` stage and no longer meets the requirements for QG 1, it passes QG 2, which means that it falls back to the `vague` stage. The same holds true for a `Model` in the `fine` stage if it no longer meets the requirements for QG 3. Then, it passes QG 4 and falls back to the `decent` stage. In the case that it also fails to meet the requirements for QG 1, e.g., confinement, it passes QG 5 and falls back to the `vague` stage. This might happen after massive changes that go beyond π_{ren} or π_{ret} operations, or when it is decided to split the `Model` into two, which then requires them to start the quality assurance process as freshly added `Models` in the `vague` stage.

Altogether, we achieve quality support for guiding model evolution in knowledge libraries (cf. `QualityLevelType` in figure 3.30), but this should never be fully automated. Hence, a modeler always needs to have the last word, as we will see later. For now, a change in stages requires confirmation of a pending indicator, making quality gates a semantic but semi-automatable concept. Still, we have clear sets of quality requirements that must be met for each stage. This counters quality as seen "'social' [... and] inherently subjective" [Moo05]. Next, we address measurement and calibration [Arb11].

Quality Measures: We designed our evolution stage automaton with quality gates to reduce some of the non-determinism that is initially required. However, we can only support some of our quality dimensions, because not all of them are measurable. Hence, we introduce properties that offer a different perspective on the characteristics that make up our syntactic, semantic, and pragmatic qualities [LSS94].

According to the degree to which the associated quality measures are inherently objective and automatable, we distinguish between strong, medium, and weak quality characteristics. The first is a fully automatable and objective means of measurement. For example, a syntax check can detect syntactic errors with absolute certainty. Further, some metrics and validators for models can provide exact quality assessments [Wüs14; GPC02; GPC05; Ste+08]. As an example, a class without a name or an attribute without a type are errors and are not only detectable by validators, but can also be helped in terms of feedback for users. We consider metamodel conformity, transformability, and defect-freeness in figure 3.26 as strong quality characteristics. The second, medium, quality characteristics have fully automatable means of measurement, but lack the ability for objective interpretation. For example, the count of classes in a model should be limited by a threshold in a knowledge library for appeal reasons. However, a count exceeding a given threshold does not automatically indicate a defect. Instead, we favor the term “smell” as a well-established notion for something that is not quite right and needs improvement [Fow99]. Hence, if a smell occurs, a modeler is notified about it but does not need to take action. Instead, the smell can be marked as ignored. To us, the medium quality characteristics in figure 3.26 are confinement, understandability, and maintainability. Finally, weak quality characteristics have neither automatable nor objective means of measurement, but heuristics can help. They can support modelers in spotting issues, but cannot provide thresholds or other guidelines. For example, `purpose` extraction can be checked by keyword extraction and comparison, but this only provides data for the modeler to make a decision. Our weak quality characteristics in figure 3.26 are semantic validity, completeness, and `purpose` extraction.

Unfortunately, this makes our quality assessment “inherently subjective” [Moo05], but we can support evaluations using reviews. We implement a system of simple reviews with five different perspectives [Bon99], called thinking hats, that have a particular focus and each disregard a reviewer’s expertise. Figure 3.28 sketches the idea and shows how five differently colored hats, symbolizing the types of review, assess a model. The main goal is to really focus on this kind of review and not to digress in other directions.

We present the short review types in alphabetical order (cf. `ReviewType` in figure 3.30 with respective `CommentTypes`): A black hat review, aka bad points judgment, is similar to the commonly known feedback, which often focuses on bad points. It requires immediate patches; hence, a high number of black hat reviews implies poor quality. A green hat review, or creativity judgment, considers alternative and new ideas as well as possible improvements. This would be one positive part of the commonly known feedback, if expressed. For `Models`, a green hat review might support a novice modeler in improving a model through inputs from a more experienced modeler who provides a green hat review. A red hat review, or emotional judgment, considers the preferences or dislikes

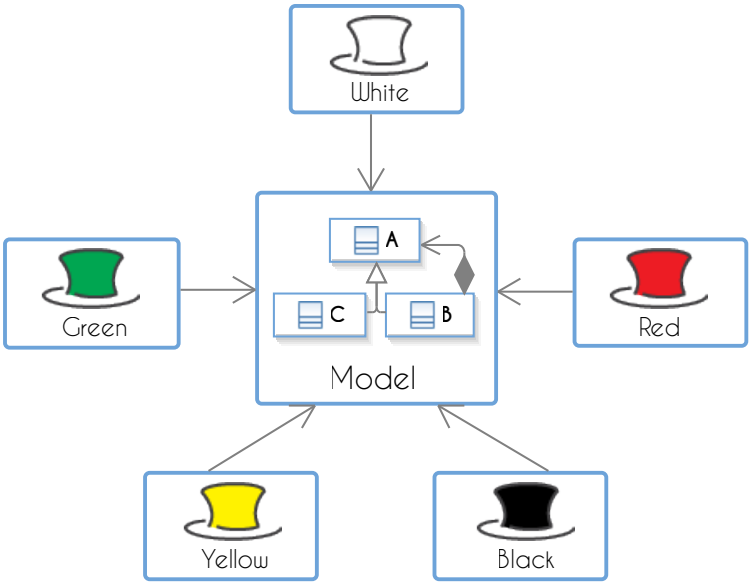


Figure 3.28.: Review Hats similar to [Gan+16]

that might result from experience. Sometimes, this is beneficial for stimulating discussion on questionable points that clearly need to be stated emotionally. A white hat review, or informational judgment, is a means of providing information not given by the model itself. At times, `Models` inherit limitations or expire for some reason, and a white hat review can retain this information. A yellow hat review, or good points judgment, is the other positive part of the commonly known feedback. This time, the information is not about possible improvements, but mentioning good points. With regard to quality, we can expect good quality if many yellow hat reviews exist.

3.4.5. Proactive Guidance for Model Evolution Stages

The quality model, quality gates, and evolution stage automaton build the foundation for our proactive quality guidance. Assessments are used during the editing process to provide feedback on a regular basis. This is beneficial for our knowledge library, because there are numerous error reports generated on request, which might be demotivating for modelers. Although this is an implementation detail, we mention some aspects here.

Our approach for proactive quality guidance comprises continuous measurement and feedback, including working instructions. The measurement triggers the calculation of metrics assigned to quality characteristics on a regular basis, so the modeler does not forget about triggering the measurements. The latter are derived instructions that are subdivided into instructions for either metrics or smells. This follows our classification of medium and weak quality characteristics and results in either clear instructions or proposals. The latter might be ignored. Note that the smell metrics may also be reported and discussed by the review hats discussed above. For the defects and smells, we only

show an excerpt of the defects from the complete list in table 3.9 [Rot13].

Table 3.9.: Excerpt of Model Defect and Smell Metrics [Rot12]

Name	Description	Reference
AttrNameOvr	Class defines a property of the same name as an inherited attribute. During code generation, this may inadvertently hide the attribute of the parent class. Consider changing the name of the attribute in the child class.	[Re04]
CyclicInheritance	Class inherits from itself directly or indirectly. The inheritance graph must be a tree; no cycles are allowed.	[Wüs14]
DupAttrNames	Class has two or more properties with identical names. Attribute names must be unique within the class.	[Wüs14]
DupOps	Class has duplicate operations. There are two or more operations with identical signatures (operation name and list of parameter types).	[Wüs14]

3.4.6. Model Evolution Stages and Generations

We have extracted several models from our running example in figure 2.3 (p. 27) so far. For example, we started section 3.2 (p. 48) by extracting three models as `Models`, namely `Airport`, `Passenger`, and `Vehicle`, and related them as shown in figure 3.4 (p. 48). In terms of versioning, they start from the same baseline, and in terms of evolution, they start in the `vague` stage. This groups them in respect of our knowledge library and in terms of evolution.

This continues our model evolution approach and extends it with the biological idea of "successive generations" mentioned at the beginning of this section [Saf14]. Figure 3.29 shows an example of a generation highlighted with a gray background made from our running example in figure 2.3 (p. 27). This has been subdivided into three models, omitting the enumerations. These three models, as `Models` in our knowledge library, form a generation in our terms because they are related by `Connectors` of syntactical type, i.e., cross-link (cf. subsection 3.3.3 and figure 3.17). Note that the models are syntactically correct and that cross-links contain "dangling" references referring to elements in models.

Moreover, cross-links between `Models` are a means of supporting chain recommendations, which raises the question of how to attribute an evolution stage to a generation. In figure 3.29, there exist several snapshots of each model, which could imply that the corresponding `Models` are in different stages. For instance, `Airport` may be in the `fine` stage while `Passenger` and `Vehicle` are in the `decent` stage. Then, the generation should be in the `decent` stage, or the lowest stage of all `Models` in a generation. Further,

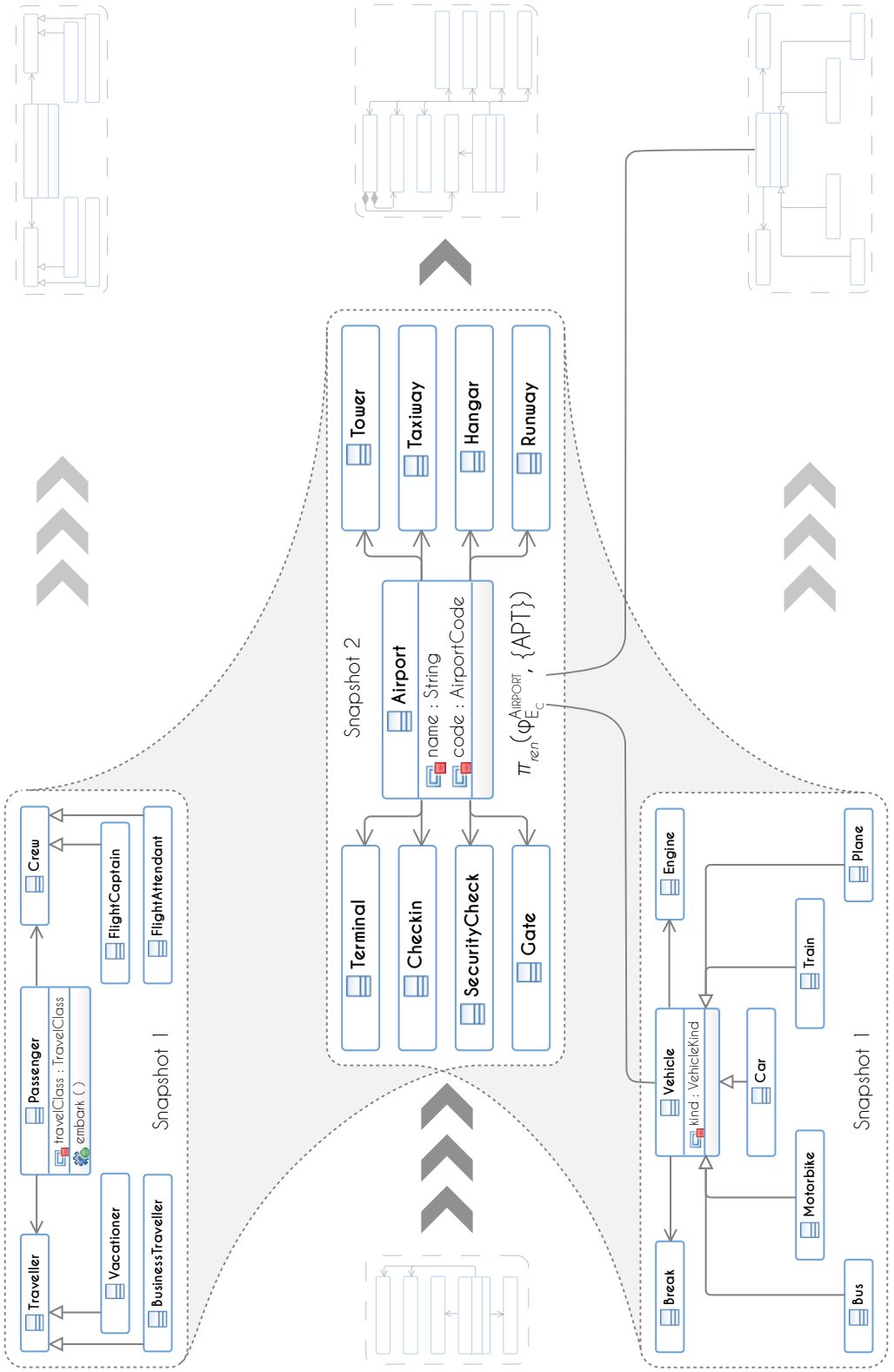


Figure 3.29.: Airport Generations (without Enumerations) [Gan+16, similar]

a generation with one `Model` flagged as deprecated should render the entire generation so, but not the other `Models`. Hence, the chain recommendations are not affected.
Impact Analysis: An altered model can lead to erroneous cross-links, so an impact analysis is required [MD00; LFR12]. Our knowledge library does not hold references to reused models, so upgrade and merge conflicts are not an issue. Further, we lock models and prevent multiple versions of one `Model`, so merging issues should not occur in our case. What remains are structural and compositional conflicts [MD00]. The former are syntactically erroneous models, leading to larger erroneous models if merged. This is handled by our quality characteristic "Defect-Freeness". Compositional conflicts occur when the merger of models leads to semantically changed models, which is part of semantic quality. Hence, syntactic and semantic consistency are taken care of [MvS05].

The remaining changes we need to investigate are additions, connections, disconnections, and removals [MD00; MvS05]. First, adding an element to a model cannot harm our knowledge library and reusability of models, because the worst scenario would be a duplicate element in an adjacent model. This potential issue requires no action because reutilization covers the case where both are reused within the same scope. The easiest solution is to prefix one of the duplicates and let the modeler decide. This also holds true for the connections case. In the case of disconnecting elements, no harm can occur because extracting one from another model could at worst introduce a smell. Altogether, the remaining conflicts result from removals, which require a little more effort to analyze.

We approach the removal of elements in models in two steps. First, we conduct a change impact analysis [BA96]. Second, we locate the impact destination, i.e., if changes need to be made to the cross-link or the adjacent model [Vác97]:

```
1  IssueSet impactAnalysis(Model model, CrossLink crossLink){
2    adjacentModel = getAdjacentModel(model, crossLink);
3    assertSyntaxOK(model, adjacentModel, crossLink);  // Error before
4    mergedModel = merge(model, adjacentModel, crossLink);
5    if (isValid(mergedModel))
6      return emptySet;                                // No harm done
7    leftLinks = modelSetDifference(crossLink, model);
8    if (!isEmpty(leftLinks))
9      return leftLinks;             // Issues found in links to model
10   rightLinks = modelSetDifference(crossLink, adjacentModel);
11   if (!isEmpty(rightLinks))
12     return rightLinks;   // Issues found in links to adjacent model
13   return emptySet;
14 }
```

Pseudocode 3.5: Change Impact Analysis

The impact analysis from pseudocode 3.5 is checked for all cross-links associated with a `Model`. Initially, we assume that the respective model and the cross-link are syntactically correct. Then, we conduct a merge test by attempting to merge the changed model with the adjacent model using the cross-link. If this results in a syntactically correct model, this change is harmless. If the result is syntactically defective, the error lies either in the cross-link or the adjacent cross-link. We can analyze this by building the set difference

of the changed model and the cross-link. If this set difference is empty, then the error must result from the adjacent model and we can locate it by building the set difference of the adjacent model and the cross-link. However, this must not result in anything other than the empty set, because the adjacent model was syntactically correct to begin with (or it was in the vague stage). Note that we can analyze the impact of this change during runtime to immediately notify the modeler.

If the impact analysis has located any issues, the change propagation needs to be conducted by the modeler. This is necessarily a manual task, because changes can alter the semantics of models and we cannot automatically fix such issues. In fact, identifiers allow for some support, even for more complex scenarios, but this is out of scope.

Example: Consider the change $\pi_{ren}(\varphi_{\mathsf{E_C}}^{\texttt{Airport}}, \{\mathsf{APT}\})$ in figure 3.29, i.e., renaming the class `Airport`. This results in an erroneous cross-link ($\mathsf{cl}^{figure\ 3.29}$), because the destination of one relationship was an `Airport` class. Given the `Airport` model (m_a), the cross-link, and the `Vehicle` model (m_v) as tuples (equations (3.1) and (3.54)) we have:

$$
\begin{aligned}
\mathsf{cl}^{figure\ 3.29} :=& (\mathsf{E_A}|_{\mathsf{m}_a \between \mathsf{m}_v}, \mathsf{E}_{cl}|_{\mathsf{m}_a \between \mathsf{m}_v}, \mathsf{E_C}|_{\mathsf{m}_a \between \mathsf{m}_v}, \dots, \rho_{\mathsf{eAttributes}}|_{\mathsf{m}_a \between \mathsf{m}_v}, \dots), \text{ with} \\
& \mathsf{E_R}|_{\mathsf{m}_a \between \mathsf{m}_v} = \{\texttt{Airport2Vehicle}, \texttt{Vehicle2Airport}\}, \\
& \rho_{\mathsf{eType}}|_{\mathsf{m}_a \between \mathsf{m}_v} = \{(\texttt{Airport2Vehicle}, \texttt{Vehicle}), \\
& \qquad\qquad (\texttt{Vehicle2Airport}, \texttt{Airport})\}, \\
& \rho_{\mathsf{eReferences}}|_{\mathsf{m}_a \between \mathsf{m}_v} = \{(\texttt{Airport}, \texttt{Airport2Vehicle}), \\
& \qquad\qquad (\texttt{Vehicle}, \texttt{Vehicle2Airport})\} \\
\mathsf{m}_a^{figure\ 3.29} :=& (\mathsf{E_A}^{figure\ 3.29\texttt{Airport}}, \mathsf{E}_{cl}^{figure\ 3.29\texttt{Airport}}, \{\varepsilon_{\mathsf{E_C}}^{\mathsf{APT}}, \dots\}, \dots)
\end{aligned}
$$

We can find a dangling element in our cross-link, because a model merge is not valid (cf. in line 5 of pseudocode 3.5) and the set difference results in:

$$
\begin{aligned}
\mathsf{cl}^{figure\ 3.29} \setminus (\mathsf{m}_a^{figure\ 3.29} \cup \mathsf{m}_v^{figure\ 3.29}) = (\mathsf{E_A}, \dots, \rho_{\mathsf{eAttributes}}, \dots), \text{ with} \\
\mathsf{E_R}^{figure\ 3.29} = \{\texttt{Airport2Vehicle}\}, \\
\rho_{\mathsf{eType}} = \{(\texttt{Airport2Vehicle}, \texttt{Vehicle})\}, \\
\rho_{\mathsf{eReferences}}^{figure\ 3.29} = \{(\texttt{Airport}, \texttt{Airport2Vehicle})\}
\end{aligned}
$$

This shows that the “Airport”, which is first in all elements, cannot be removed, because it is not in the union set of m_a and m_v. Note that the undefined sets, e.g., $\mathsf{E_A}$, are empty.

With operation-based cross-links from equation (3.60) (p. 86), we can also demonstrate: Given `Airport` and `Vehicle` in operation-based form ($\Im_{Ecore}^{\texttt{Airport}}$, $\Im_{Ecore}^{\texttt{Vehicle}}$), we obtain:

$$
\begin{aligned}
\Im_{Ecore}^{figure\ 3.29} =& \pi_{rass}(\varphi_{\mathsf{E_C}}^{\texttt{Vehicle}}, \{\varphi_{\mathsf{E_R}}^{\texttt{Vehicle2Airport}}\}) \circ \pi_{rass}(\varphi_{\mathsf{E_C}}^{\dagger\texttt{Airport}\dagger}, \{\varphi_{\mathsf{E_R}}^{\texttt{Airport2Vehicle}}\}) \circ \\
& \pi_c(\mathsf{E_R}^{\texttt{Airport2Vehicle}}) \circ \pi_c(\mathsf{E_R}^{\texttt{Vehicle2Airport}}) \circ \\
& \pi_{ren}(\varphi_{\mathsf{E_C}}^{\texttt{Airport}}, \{\mathsf{APT}\}) \circ \Im_{Ecore}^{\texttt{Airport}} \circ \Im_{Ecore}^{\texttt{Vehicle}}
\end{aligned}
\tag{3.70}
$$

We can see how two `Models` were created and a renaming took place in the last line of equation (3.70). The references were created in the penultimate line and, after that, the assignment fails, because $\varphi_{\mathsf{E_C}}^{\texttt{Airport}}$ cannot be resolved in the first line, as indicated by †. Nevertheless, we could rewrite pseudocode 3.5 by means of find operations. We omit this for the sake of brevity, but mention that this view is particularly useful for a knowledge library with support for hyperedges.

3.4.7. Design Rationales and Observations

The approach for model evolution, which we introduced above, directly addresses the "Evolution Challenge" discussed in section 1.2 (p. 6), and is intended to support reuse, as we explain in section 3.5 (p. 117). Hence, the major goal was an approach in line with our project goals, i.e., the E and S denoting "easily and seamlessly" in HERMES.

Our initial thinking while designing an easy model evolution approach was that satisfied customers are those who come back, and that quality makes a large contribution to customer satisfaction. Nevertheless, our harvesting, which must also be easy, does not overburden modelers with quality concerns. Hence, we decided to place quality concerns in a separate, successive task. This way, we avoid annoying modelers with lengthy mining operations, which potentially leave models in our knowledge library with quite some potential for improvement. At the same time, we are cautious of changing the requirements of models that are in our knowledge library. This means, even if a perfectly quality assured model makes it into our knowledge library, it is likely to face "aging" without change, a phenomenon observed in software [Par94]. As a result, our thinking in iterations, i.e., "first we do things, and then we do them right", makes perfect sense, and we first place models in our knowledge library and then start quality assurance. In other words, we accept that requirement changes happen to models in our knowledge library and support them independently of deployment scenarios, while allowing and enabling feedback enhancement loops.

With this separation of concerns established, we seek an easy solution for quality assurance in terms of comprehensibility. Therefore, we root our three stages, which eventually aggregate to quality statements, in the traffic light metaphor with commonly known colors and exactly three stages. This also fosters simple filtering regarding user expertise, i.e., novice users can only use models in the `fine` stage, more advanced users can also use models in the `decent` stage, and expert users can use all models. Simply put, experts are the jaywalkers in our approach, because they might need to work with defective models.

Now the question arises whether fewer or more stages could have sufficed and we tested that but could not come up with an alternative of appropriate complexity [Rot12]. Discussing this, we need to keep in mind that the stages are not the only complexity contributing to cognitive load concerning our model evolution approach. There are review hats, quality attributes, and other concepts as well.

This raises the question of their complexity and why we designed all these concepts as we did. Firstly, the review hats are those proposed by Bono, because our testing did not

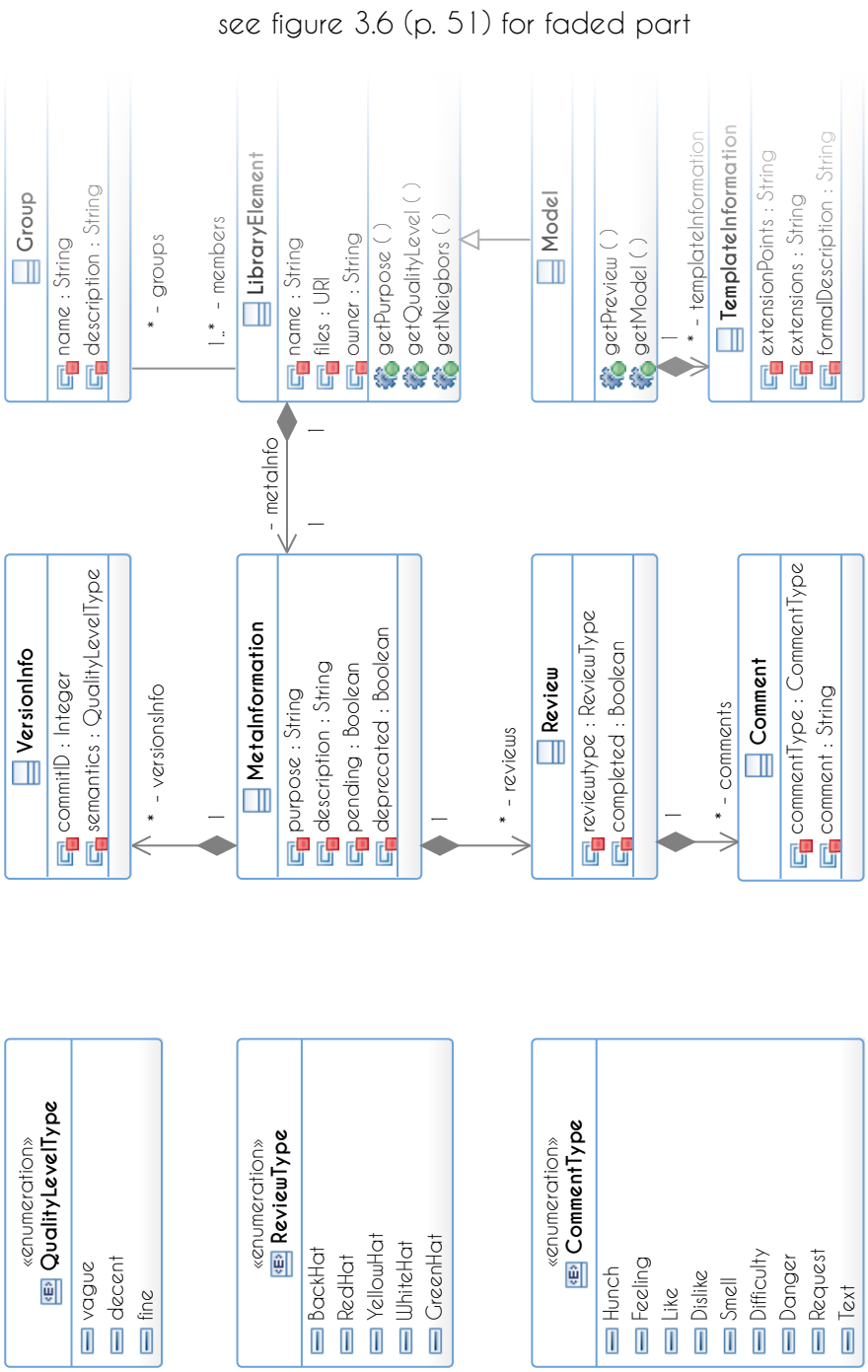


Figure 3.30.: MDF: Knowledge Library Extension (cf. [GL13] complete on page 200)

require us to alter an established approach [Rot12]. Secondly, our quality model describes a basis, as we explained above, but in discussing this, it is more important to ask why we did not provide a tailoring mechanism similar to profiles for bi-directional quality models [Hil10]. Certainly, there is room for bi-directional quality models, because the metrics that we gather could be assessed quite differently from our approach. This induces another dimension of complexity, because we require an abstraction level that maps subjective perception of quality via so-called indicators to metrics [Sch10]. Further, a mechanism similar to using profiles, i.e., requiring only a few of the quality characteristics of a quality model, can be appropriate. However, that is exactly what our stages represent and gates ensure. Hence, without going into detail, we use this idea without explicitly referring to it. There has been extensive research on quality models in computer science, and we restrict ourselves to pragmatism in this respect. Thirdly, we need to address our deprecation flag and why we did not introduce it as another stage. This is an attempt to persist knowledge in the final stage. This explicitly keeps the last stage and only marks a model as out of service. Hence, it can still serve several purposes, e.g., alternative or historic examples. In fact, any production should document the reasons for deprecation and the alternatives that replace the deprecated model. Put differently, this makes our stages include a deprecation flag that is very similar to software release cycles, from `vague` all the way to `fine`, which a recommender can use to become a so-called persuasive recommender, i.e., provide information about why something is recommended [YGZ13]. This not only holds true for, e.g., matching terms as rations, but also for counterexamples for historic reasons. Finally, our evolution stage automaton and the quality gates have this kind of formality because we eventually require code generation (cf. section 4.4 (p. 155)). Further, our line of discourse requires some concepts that eventually (mostly) disappear, e.g., the non-determinism in our automaton.

As another attempt to lower complexity and further enhance usability, we designed proactive quality guidance as a means of easing quality assurance. We will pick up on the idea of proactive support while discussing user interaction in subsection 3.5.3 (p. 122) for model recommendation production. In addition, we will discuss operation-based models and how they fit into the bigger picture.

For reasons of clarity, we omit some aspects from our approach. First, we omit comment types in figure 3.30, but do not explicitly encompass them in our explanations; in fact, they are enclosed with each review. Reviews imply `CommentTypes`, and this is encouraged by our GUI. We omit a more detailed discussion because the derived types are intuitive and this approach makes no conceptual difference, but helps users. Second, we have neither roles nor user management implemented in the approach or realization. As a result, any user can change any stage and boost a freshly added model to the `fine` stage with a few clicks. We consider this beyond the scope of this approach, and keep track of the users who change anything by adding a name field to be filled in. Finally, we do not consider the model metrics in our contribution, so we provide no more than a reference to what previous methods are employed. Indeed, this part is designed and built to be exchangeable, as indicated in the explanations concerning figure 3.23. The same holds true for the versioning underlying this approach.

3.4.8. Related Work

Research on evolution and model evolution has been conducted for some time, so we first examine some meta-research with the intention of classifying our approach. First, "the four different dimensions of evolution in model-driven development" by van Deursen, Visser, and Warmer places us in *regular evolution* [vVW07]. Note that we cannot be categorized as abstract evolution, because our knowledge library is detached from the reused models and change propagation in that direction is not under consideration. Second, "the two orthogonal model-evolution dimensions" proposed by Biehl would classify our approach as *content-related changes* and *local evolution* [Bie10]. Our framework is local, because we do not take the abstract syntax into account, and we are content-related because we isolate ripple effects and do not propagate changes between related models. Third, regarding "the three types of changes in software model evolution" by Levendovszky et al., we deal with *requirements* and *style* [Lev+11]. The former results from our purpose and its extraction, and the latter comes from our quality characteristics regarding understandability and maintainability. Other types of changes we deal with are *corrective* and *perfective*, as proposed by Swanson [Swa76].

Bearing these classifications in mind, we can contrast our approach with other research regarding model evolution, quality in modeling, and change propagation.

Model Evolution: The evolution of software artifacts goes back to Lehman [Leh80], and Godfrey and German state that this reaches back even further [GG08]. A more recent research roadmap was provided by Bennett and Rajlich [BR00], but there was little to no research on evolution for knowledge libraries comprising models.

Model evolution is usually researched with "evolution as a goal" that needs to be automated, e.g., in tools for model-driven development. The first related approach is called COPE and was presented by Herrmannsdörfer [Her11]. It deals with model co-evolution in an operation-based way and uses editing traces in sequences, as in our framework. Further, these sequences can be held in central repositories, so they can be forwarded to other models. In a sense, this puts model migration in focus, and this best explains the difference from our approach. Herrmannsdörfer provides more formally defined operations derived from experiences or gained from object-oriented database-schema migrations and object-oriented source-code refactorings. In more detail, some high-level operations are refined to structural and non-structural primitives, which make up a catalogue. Examples are operations for specialization and delegation, or operations for replacement and merge-split [HVW11]. We use CMOF API [Obj14], but many ideas could be beneficial in our approach.

The second related approach is called MoDisco [Ecl14b]. Currently, this is jointly hosted with AM3 and provides model management functionality [All+06]. However, the roots of MoDisco lie in support for the evolution of legacy systems by means of model-driven development. To that end, MoDisco supports re-engineering and co-evolution. Our approach differs in the sense that we support guided evolution with quality assessments.

SiDiff, developed by Wenzel, Hutter, and Kelter, traces model elements in version control systems [WHK07]. Hence, the authors put their approach in the category of repositories, but SiDiff can monitor model changes to find commonalities and differences.

This is almost identical to our ideas about snapshots and evolution steps, but unlike us, Wenzel, Hutter, and Kelter do not research staging and quality guidance.

Sun, White, and Gray take an alternative perspective for model evolution, considering model transformation by demonstration with a tool called MT-Scribe [SWG09]. Here, demonstrations are a means of supporting users in finding model-evolution tasks automatically [Tay+11]. Hence, a user who is not a modeling expert demonstrates a change, and MT-Scribe derives tasks from this demonstration while keeping in mind model-evolution. This is fundamentally different to our approach, because reuse is not in focus, but results in end-users seeing evolution in small, unguided steps.

Quality in Modeling: Our approach profoundly relies on quality in modeling, and we are aware of difficulties beyond those discussed at length by Moody [Moo05]. Therefore, we tried to overcome the discussed subjective manner by subdividing quality into a syntactic, semantic, and pragmatic dimension, as proposed by Lindland, Sindre, and Solvberg [LSS94], while keeping in mind that these dimensions might interfere with each other, as explained by Bansiya and Davis [BD02].

Many quality models exist for UML modeling, and we base our work on Lange's contributions with additions from McQuillan and Power and Kim and Boldyreff [Lan06; MP06; KB02]. Further, we considered work by Genero et al., Wedemeijer, and Mohagheghi and Dehlen for our metrics catalog [Gen+03; Wed01; MD09]. In particular, Mohagheghi and Dehlen provide the linkage between quality characteristics and metrics for special goals, i.e., purposes, which was beneficial for our metrics catalog.

In addition, Mohagheghi and Dehlen mention challenges in measurement [MD09]. We approach these challenges, considering that evolution in a knowledge library eases them already, with the Six Thinking Hats of Bono, which we adapted to our needs [Bon99].

Regarding implementations, the MetricViewEvolution software is the most similar we have found [LWC07b]. This was developed by Lange, Wijns, and Chaudron and offers different views on evolving models. The six views illustrate context, evolution, meta, metric, UML-city, and quality. This separates information as a single concern, hiding other conceptual details. The approach itself is presented by Lange, Wijns, and Chaudron [LWC07a]. We do the same with review hats for concepts and lightweight UIs without dedicated views on certain aspects.

The EMFMetrics implementation, by Arendt, Stepiena, and Taentzer [AST10] and now merged with EMF Refactor by Arendt, Mantz, and Taentzer [AMT10], provides a measurement environment for modeling [AT13], which we also use. This offers model metrics and defines model smells, which now became a reasoning platform for refactorings [AT12]. In contrast to EMF Refactor, we offer edit-time assessment and provide more detailed means of guiding changes and reviewing defects, smells, and hunches.

Change Propagation: We do change impact analysis and propagation with changes in model generations. This is different from the ripple effects in model-consistency management, model-change propagation, or model synchronization [PvM15], because our approach is local and content-related, i.e., our cross-links serve as borders for change propagation. In terms of Levendovszky et al., we employ intra- and inter-model changes in a single modeling domain [Lev+11]. For more generic software change type foundations,

consult Lehnert, Farooq, and Riebisch [LFR12].

An approach that adapts ideas from reuse contracts to evolution contracts is that of Mens and D'Hondt [MD00]. This extends the UML metamodel using a mechanism for disciplined change tracking and guidance [Obj11b]. Further, the impact analysis, change propagation, and consistency verification are taken into account by the "framework for managing the consistency of evolving UML models" by Mens, van der Straeten, Ragnhild, and Simmonds [MvS05]. This ensures "safe model evolution", but for us, an extension of Ecore was not an option.

Odyssey-VCS by Murta et al. implements mechanisms from source-code and version-control systems in models by detecting change traces through model-mining [Mur+07; DMW05]. The approach is presented by Dantas, Murta, and Werner and the realization of Odyssey-VCS identifies changes in several diagram types and link traces. Compared to our approach, the model-mining techniques are not necessary, because we perform edit-time monitoring for operations.

Regarding the "consistency between UML models", Mens, van der Straeten, Ragnhild, and Simmonds use description logics [MvS03]. They take XMI exports, transform them into their representation in description logics, and distinguish between "horizontal and evolution consistency". These regard consistency between models of the same version and between versions of one model, respectively. Hence, we consider horizontal consistency with our generations, but, as we showed in our example, we do not require the elements of our generation to be from the same version.

Briand et al. studied the "automated impact analysis of UML models" [Bri+06] and proposed a change taxonomy and a methodological framework. Their realization is founded in OCL, so the approach is based on a formal foundation given for model changes, change impact, and "bag[s] of impacted elements". These are basically sets of elements impacted by changes. Further, they develop distance measures between changed and affected elements, all of which is automated. For us, a realization in OCL was not an option, and our solution could not be fully automatable for semantic reasons.

The final domain that can be regarded as related is model refactoring, for which Boger, Sturm, and Fragemann provide an overview [BSF03]. An example of such an approach using OAW/OCL queries is presented by Enckevort [Enc09], and Astels provide an approach for larger systems [Ast02]. Also in this category is the "Operation Recorder" developed by Brosch et al. [Bro+09]. This subdivides a set of refactorings into eight steps, starting with "creating the initial model" and proceeding until the "generation of a specific artifact". They demonstrate their approach with an example dealing with state machines. Their work on conflict detection is similar to ours, but their domain differs, so we have only picked up certain ideas.

3.4.9. Summary of Evolving Models

The previous section introduced an evolution concept for knowledge libraries. We defined snapshots (S) of models (cf. top of figure 3.31) and how sequences of operations perform evolution steps (σ at the top of figure 3.31). These, as alternating sequences of model

snapshots and evolution steps (S_i, σ_i, S_j, σ_j, S_k, σ_k, ...) represent our understanding of model evolution. However, the alternating sequences can be looked at as model evolution graphs, and we defined vertices as a set of snapshots (S), with the connecting edges made of pairs of snapshots (S_i, S_j). Further, we introduced two labeling functions for assigning snapshots to identifiers and edges to evolution steps (σ), i.e., sequences of operations. This allowed us to introduce evolution stages ($_{-,\circ,+}$ subscripts throughout figure 3.31) for labeling the reusability of models as low, medium, or high (low also means sketchy, medium also means provisional, and high also means stable). As a formality, we introduced an evolution stage automaton denoting the low stages as vaguely reusable, medium as decently reusable, and high as finely reusable. We then added quality gates (QG in figure 3.31) to the evolution stage automaton (cf. bottom right in figure 3.31) to prepare proactive quality guidance. We achieved this by assigning the quality characteristics of a quality model (cf. left of figure 3.31) to the quality gates, which serve as semantic checkpoints. Then, we implemented the exemplary quality characteristics with weak, medium, and strong quality measures. The latter are non-negotiable errors that need fixing, e.g., syntactical errors. The medium measures are negotiable metrics with thresholds that should not, but can, be exceeded. Finally, the weak quality measures are of semantic nature and their measurement is neither automatable nor objective, so we provide a short review mechanism based on review hats to assess these. In addition, we declared a deprecated flag to indicate when a `Model` had become out of service without losing its stage. Finally, we looked into generations of models, how to determine their degree of reusability, and how to conduct change impact analysis.

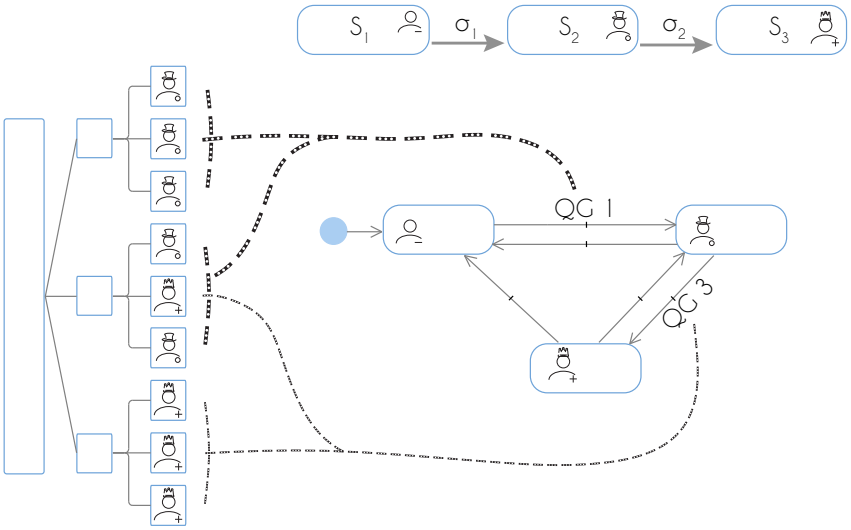
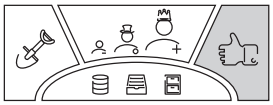


Figure 3.31.: Evolution Example Summarizing figures 3.22, 3.26 and 3.27

3.5. Reusing Models



Content-assist systems and other completion mechanisms are built into almost every state-of-the-art IDE. They allow developers, or more general users, to have information at their fingertips that would otherwise be woven into the complexity of tasks and their environment. In programming, that could limit the possible completions by taking into account what has already been typed as a prefix. By reducing such potential information overflow and guiding users, IDEs reduce the cognitive load and improve quality by preventing typographic errors.

However, complex systems induce manifold completion options that are not chosen equally often. In popular examples, several hundred options are available for a programmer to choose from [Ecl14a], but few of these options are actually picked. At least, this is what a recommender system learned from studying available source code. This means that an approach already successfully applied by online shops [LSY03], video streaming services [BL07], and Internet searches [SCB11] was successfully transferred to IDEs.

On a more general note, these recommender systems for software engineering (RSSEs) help unveil or discover previously unknown information [Wal13]. Thus far, the research areas comprise requirements engineering [Fel+13], programming [Mas+05; WKB09], predictive component selection [MLA04], and issue management [BR14]. Altogether, efforts are being made to ease information overflow in project landscapes for present or new members, particularly for knowledge-intensive tasks [Dag+10].

Another knowledge-intensive task is modeling, but efforts at transferring ideas from recommender systems to modeling activities as a means of approaching the retrieval challenge discussed in section 1.2 (p. 6) have been sparse [DGL14a]. We have developed a knowledge library that is capable of providing models, but it remains to produce meaningful recommendations. Note that we address reuse foremost, because “[e]fficient reuse must be done within a higher level of abstraction; [since] the design level is more important then[sic] the coding one” [Gom+04]. Further, because we aim for “good models”, we commit to “the importance of ‘good’ models from the beginning” [GPC05].

How is it possible to produce model recommendations? Given our knowledge library ($\mathbb{KL}^{\textit{figure 3.7}}$) from equation (3.38) (p. 61), we can, initially, think of several cases for recommending the `Airport Model` ($\varepsilon_{\mathbb{M}}^{\texttt{Airport}}$) sketched in figure 3.32. First, an explicit Airport `Query` ($\Phi_{\mathbb{KL}^{\textit{figure 3.7}}}^{\textsc{queryterm}}$) could gain this `Model`. Second, an already-present `Airport` class could automatically initiate this result. Third, an already-present part similar to our `Vehicle Model` could determine ($\varepsilon_{\mathbb{M}}^{\texttt{Airport}}$) as a neighbor in our knowledge library and provide it automatically. Fourth, a just-placed `Passenger Model` could lead to an automatically produced follow-up. Altogether, we can sketch the resulting scenario, as illustrated in figure 3.32, denoting the chosen `Model` by a “thumbs up” icon, 👍, and the rejected `Models` by faded text and a border:

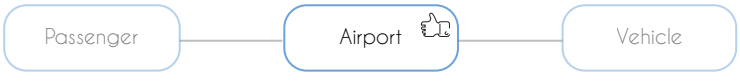


Figure 3.32.: Recommendation Example for `Airport` from figure 3.4 (p. 48)

In fact, this example only scratches the surface of the possibilities, as we will elaborate after we have laid the necessary groundwork. So far, we can deduce from our examples regarding figure 3.32 that a QUERYING and an editing context might be involved in producing recommendations from our knowledge library. Further, relationships in this knowledge library and the granularity of elements are directions to explore, as explained above. This also encompasses compound elements or parts of them. Subsequently, we elaborate on our experiences and previous work [DGL13; DGL14a; DGL14b].

3.5.1. The Model Recommender Framework

Our examples regarding figure 3.32 as well as our experience and explanations above already indicate that a *model recommender framework* comprises several parts [DGL14a]. These are depicted in figure 3.33. They show the *UIs*, which could be triggered either explicitly (reactively) by a user action or implicitly (proactively). Further, the parts show that many *contexts* are present, and these could contain editing traces, IDE canvas analysis and transformations, or project specifics. Finally, multiple *recommender strategies* are shown in figure 3.33. Each could leverage different data for inputs or data sources for producing recommendations. We elaborate on details of all parts, but first outline the foundations.

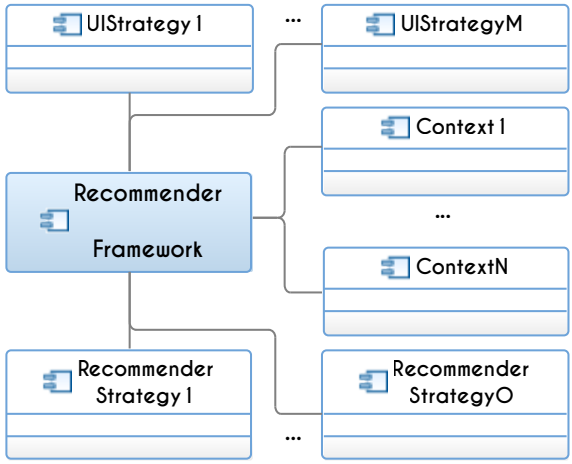


Figure 3.33.: Model Recommender Framework (MRF) similar to [DGL14a]

3.5.2. Model Recommendations, Operations, and Dimensions

At the heart of our subsequent discourse are model recommendations and candidates. We build these terms using the ideas introduced in subsection 2.3.2 (p. 19) for general recommender systems, and adjust them with a term built for RSSEs. In doing so, we agree that, even after years of discussion, the following definition is “the most useful for distinguishing RSSEs from other software engineering tools” [RW14]):

> An RSSE is a software application that provides information items estimated to be valuable for a software engineering task in a given context [RWZ10].

This definition sets our roadmap for approaching model recommendations and respective systems, because it comprises the essential required parts. These parts also map to figure 3.33 as follows: First, our "engineering task" is modeling pursued by UIs; we limit this to UML class diagrams, because we investigate conceptual modeling. Note that the focus is on the "task", which makes RSSEs "task-centric, as opposed to the user-centric nature of traditional recommendation systems" [RW14]. This limits the other aspects from the definition accordingly. Second, the "given context" subsumes information that is indirectly relevant for data processing by the "software application", and we can consider this context as the immediate environment or the related surroundings, as explained above. This means that, third, the effective "software application" can be viewed as the entire figure 3.33, and we need only explain what we alluded to as the heart above.

This heart is intended "to provide information items [] valuable" for modeling, which we can subdivide into three parts. First, "to provide" information items is the responsibility of recommender strategies in figure 3.33, and they are meant to do so by means of evaluating the contextual information, i.e., attempt to provide, second, "valuable" items. This means they need to expose characteristics for evaluation such as "(1) novelty and surprise [...] and (2) familiarity and reinforcement" [RW14]. We will come back to this in subsection 3.5.6. Finally, "information items" are model recommendations for us:

> A model recommendation (item) is a ranked and valuable information item produced by an RSSE for modeling given constraint dimensions.

Neglecting dimensions for a moment, producing model recommendations can be subdivided into four recommendation operations, which work on model recommendation candidates [LGS11]. We summarize the operations as a set denoted by a capital ϱ (P) in equation (3.71). Given a knowledge library ($\mathbb{KL} \in \mathcal{KL}$), they are a model ($m \in \mathcal{M}$) as part of a context ($c \in \mathcal{C}$) and a QUERYTERM ($\in$ STR), and are described as follows: First, the provided data are analyzed (ϱ_{ana} equation (3.72)). Second, candidate sets are generated (ϱ_{gen} equation (3.73)). Third, elements of these candidate sets are ranked (ϱ_{rnk} equation (3.74)) and finally filtered (ϱ_{fil} equation (3.75)) to become model recommendations, i.e., results. We dedicate subsections to these operations and introduce them in detail in equations (3.72) to (3.75).

$$\mathrm{P} := \{\varrho_{ana}, \varrho_{gen}, \varrho_{rnk}, \varrho_{fil}\} \qquad \textbf{Recommendation Operations} \qquad (3.71)$$

$$\varrho_{ana} := \mathcal{KL} \times \mathcal{C} \times \mathrm{STR} \rightarrow \mathcal{KL} \times \mathcal{C} \times \mathrm{STR} \times \mathcal{MRC} : \qquad \text{subsection 3.5.4} \qquad (3.72)$$

$$\varrho_{gen} := \mathcal{KL} \times \mathcal{C} \times \mathrm{STR} \times \mathcal{MRC} \rightarrow \mathcal{KL} \times \mathcal{C} \times \mathrm{STR} \times \mathcal{MRC} : \qquad \text{subsection 3.5.5} \qquad (3.73)$$

$$\varrho_{rnk} := \mathcal{KL} \times \mathcal{C} \times \mathrm{STR} \times \mathcal{MRC} \rightarrow \mathcal{C} \times \mathcal{MR} : \qquad \text{subsection 3.5.6} \qquad (3.74)$$

$$\varrho_{fil} := \mathcal{C} \times \mathcal{MR} \rightarrow \bigtimes (\mathrm{E}_{\mathrm{M}} \times \mathbb{R}^{+}) : \qquad \text{subsection 3.5.6} \qquad (3.75)$$

Note that ϱ_{gen} has an identical domain and co-domain. This allows successive ϱ_{gen} operations, which can take into account the provided model recommendation candidates ($\mathcal{MRC}$ cf. subsection 3.5.5) and, e.g., consider neighbors. Accordingly, our analysis

operation (ϱ_{ana}) has an adjusted co-domain with an additional set for model recommendation candidates, which, eventually, comprises a tuple of empty sets. Further, note that the domains and co-domains in P induce a processing order. Combining these aspects, we can denote model recommendation production as MP in equation (3.76) and explain the production sequence of model recommendation operations as shown in equation (3.76). The result is a set of tuples made of models and their respective ranks:

$$\begin{aligned} \mathsf{MP} := \mathcal{KL} \times \mathcal{C} \times \textsc{str} \to \bigtimes (\mathcal{M} \times \mathbb{R}^+) : \\ (\mathbb{KL}, \mathsf{c}, \textsc{queryterm}) \mapsto \varrho_{fil} \circ \varrho_{rnk} \circ \bigcirc_{i=0}^{n \in \mathbb{N}} \varrho_{gen_i} \circ \varrho_{ana} \end{aligned} \tag{3.76}$$

For the sake of simplicity, we introduce a shortcut notation for this model recommendation production operation (MP). Similar to our find operation in models (φ equation (3.14) (p. 39)) or `Query` operation for knowledge libraries (Φ equation (3.41) (p. 63)), the subscript provides the scope and the superscript denotes the query term. In addition, the middle holds the contextual information. For a given $\mathbb{KL} \in \mathcal{KL}$, we can write:

$$\mathsf{MP}_{\substack{\mathsf{c} \\ \mathbb{KL}}}^{\textsc{queryterm}} := \mathsf{MP}(\mathbb{KL}, \mathsf{c}, \textsc{queryterm}), \qquad \mathsf{c} \in \mathcal{C},\ \textsc{queryterm} \in \textsc{str} \tag{3.77}$$

Given this understanding of model recommendations, we can turn to constraint dimensions and their affected properties, omitting their structural nature for a moment. The dimensions stretch in three directions, and we discuss each in terms of the limitations they impose on model recommendation production. We can regard these limitations as degrees of freedom for "how a recommendation could be produced with these properties" or "which turns an algorithm might take to yield this recommendation". Altogether, this frame helps structure model recommendation production in subsection 3.5.5, granted that a context as in figure 3.33 delivers additional data and a recommender strategy as in figure 3.33 can rely on a knowledge library.

The dimensions of the constraints are user interaction, available data, and permitted scope, as listed in table 3.10. The first might lead to different model recommendations, because explicit (*reactive*) triggering is task-intrusive, whereas implicit (*proactive*) triggering is not. Similarly, GUIs might limit the *extent* of the presented or preview-able recommendations [DGL14b]. We elaborate on UI and GUI considerations in subsection 3.5.3. Next, available data allow for the production of more precise recommendations, as they enable more specific querying and feature leverage. Whereas simple querying might rely on search terms, contextual information can enhance the *sensitivity* (precision) of model recommendation production. For example, an active model or editing sequence can provide beneficial data, as we show in subsection 3.5.5. Further, a context analysis, which might include information from a requirements document, could provide further sensitivity enhancements. Finally, the *permitted scope*, which is allowed or taken into account, changes the results. Consider a query that results in only one recommendation. This could be extended, and also comprise a list with neighbors (cf. table 3.11), which are `Models` linked via a `Connector`, e.g., `Passenger` or `Vehicle` in figure 3.32.

Table 3.10.: Constraint Dimensions of Model Recommendation (Item) Production

Constraint	Spread	Affected Property	Explanation
User Interface	pro-/reactive trigger	extent	see table 3.11
Available Data	context, library	sensitivity	extend of data used
Permitted Scope	immediate, implicit results	granularity relatedness impact	see table 3.11 see table 3.11 extending, ... deleting

The dimensions introduced above affect the model recommendation properties, namely, sensitivity, impact, and extent. We subdivide the latter into *granularity* and *relatedness*. Altogether, we can look at these properties, which reappear during model recommendation candidate generation in subsection 3.5.5, as follows: For a given context (sensitivity), we query our knowledge library (granularity, relatedness), generate model recommendation candidates (extent), and possibly apply them (*impact*). In other words, we can see the freedom of model recommendations on scales (de-)limited by granularity and relatedness, (un-)deduced by sensitivity, and (un-)filtered for extent (cf. Configuration in appendix A (p. 195)).

Table 3.11.: Model Recommendation (Item) Extent: Granularity and Relatedness

Name	Gra.	Rel.	Explanation	Example
element	✓		element of model (m)	table 3.3 (p. 37)
submodel	✓		part, submodel (s) of a model (m)	equation (3.47) (p. 75)
complete	✓		model (m) as is	equation (3.1) (p. 36)
chain	✓	(✓)	`Connector` with(-out) cross-link (cl)	figure 3.17 (p. 83)
grouped		✓	`Group` between `Models`	figure 3.7 (p. 53)
categorized		✓	`Category` between `Models`	figure 3.7 (p. 53)

In more detail, the first property, which we call sensitivity, relates to how much a context is involved while producing a model recommendation. Whether the edited model, an editing sequence, or other aspects, which are not immediately related to a query term, are taken into account for producing recommendations impacts on this property. Hence, we could perceive this as a context-sensitivity of model recommendations. Second, relatedness expresses the degree to which the environment of an item in our knowledge library is considered as an option for generating model recommendation candidates. Imagine a `Model` from our knowledge library considered as a model recommendation candidate. Further, imagine this `Model` with adjacent `Models` linked by `Connectors`, grouped by `Groups`, or categorized by `Categorys`, e.g., $\varepsilon_{\mathsf{M}}^{\texttt{Airport}}$ in figure 3.7 (p. 53).

Relatedness now expresses the degree to which the adjacent or related `Model` should be considered as additional model recommendations candidate. This also means that relatedness is determined by the permitted scope, as introduced above. Third, granularity expresses the opposite of relatedness, i.e., not aspects provided by a knowledge library, but the syntactic content of a model. Hence, the smallest elements in granularity are the model elements of sets introduced in table 3.1 (p. 34), and larger parts are the submodels explained in equation (3.47) (p. 75). Additionally, we consider entire models and even joined models, e.g., built by two models and a cross-link, as a granularity. In other words, we speak of element, submodel, complete, and chain granularity. This is also summarized for relatedness and granularity in table 3.11. Finally, impact is a property related to the expected change in a model recommendation if reutilized. It can range from idempotent, i.e., non-altering, to introducing, i.e., adding new content. In between lie the attributes of extending and altering. The latter is considered as a form of restructuring, which is often called "refactoring" [SU13; KM14]. In addition, the impact of a model recommendation could also restrict or delete information. We mention this just for completeness and do not go into further detail, because model recommendations should provide valuable information, not just restructuring.

3.5.3. User Interface Considerations

With an understanding of model recommendations and their granularity established, we can discuss UIs based on guidelines for recommender systems and their delivery in more detail [Pu+11; MM14]. These cover the topics of an interaction-model, triggering, and representation limitation. Many of these can constrain model recommendation production, as we have introduced above.

We can apply common interaction models to provide a bigger frame for our model recommendation operations from equation (3.71) in figure 3.34 [Pu+11]. As a first step, an initial preference specification is undertaken. In our case, this is an entered QUERYTERM, e.g., in equation (3.77), but it can also include additional contextual information. In the next step, data processing takes place. This means that a production sequence, as shown in equation (3.76), is performed and yields a set of recommendations. In the third step, either a recommendation is picked from the set or the initial preference specification is revised. In short, the interaction is: specify preference, produce recommendation result set (MP), either pick recommendation or revise preference [Pu+11].

What has not been considered in this observation is the initiation of this interaction, which can be proactive or reactive. The former takes place if "[recommendations are presented when deemed appropriate]", i.e., automatically, and the latter occurs if "[recommendations are presented only when requested]", i.e., manually [Rob+14, both Glossary]. For reactively triggered interactions, this means that a search field could be filled in and an action button explicitly triggers the interaction. However, for a proactive interaction, the question arises of how the system determines the "appropriate" time. Possible answers are timeouts or a completed editing. An example for the latter could be a created class or a selection of classes [DGL14b]. This leads to the interaction model shown in figure 3.34.

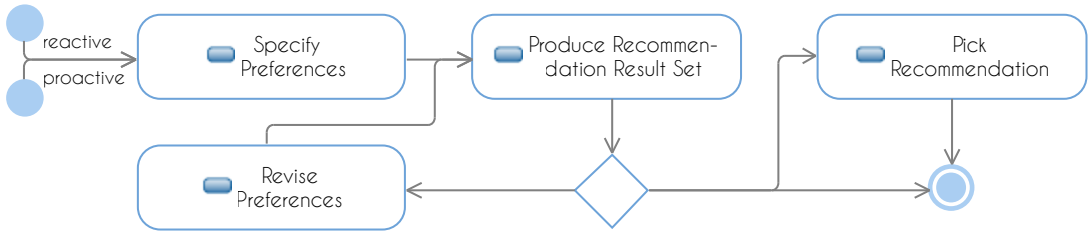


Figure 3.34.: Recommender Framework Interaction Model adopted from [Pu+11]

In discussing this interaction model, we do not consider alternative production sequences of model recommendation operations or multiple concurrent ones. In the former case, interaction remains the same, because we can consider variants of equation (3.76) as black boxes. This is different for multiple and concurrent production sequences of model recommendation operations if they are not joint before completion. Then, the interaction model changes marginally, because concurrent recommender strategies that started simultaneously but finished separately require an altered presentation. This presentation needs to keep up with successively delivered recommendations and rearrange them as new recommendations [DGL14b]. This imposes an alteration on UI implementations and makes it necessary to monitor the recommender strategies, as we illustrate in figure 3.37 in subsection 3.5.5.

Other UI aspects that are relevant for user interaction occur at the graphical design level. They deal with different graphical querying options such as search boxes or preview functionality such as overlays. These place limitations on interactions and GUIs in terms of screen space constraints or comprehensibility. The impacts are interrelated, because more extensive and complex graphical representations are often more difficult to comprehend. This is counterproductive for an “easy” reuse approach. We omit further details of our previous work for brevity [DGL14b].

3.5.4. Recommendation Context and Analysis

Data processing is often context sensitive and the results depend on contextual information provided as inputs. The research area of “context modeling and reasoning” focuses on these in respect of environmental contextual information, i.e., raw sensor data [Bet+10]. Combining this with recommender systems leads to location-based services, which fall into a broader category called, second, “context-aware recommender systems (CARS)”. We can consider this as the initiation of contextual analysis for recommender systems, the value of which is often underestimated [AT11]. At the core of CARS are three-dimensional prediction functions similar to our MP operation. However, introducing model recommendations, we find that they are task-centric rather than user-centric activities. Thus, contextual information for software engineering tasks comes with different characteristics. This guides us to the third research area that is often found in source-code related environments, e.g., programming in Eclipse supported by Mylyn

[KM05; KM06]. Subsequently, we bring these areas together and explain our universe of contexts ($\mathcal{C}$), its dimensions, and our context analysis operation (ϱ_{ana} in equation (3.72)).

Various definitions of the term context have been given, and 150 of these were analyzed in more detail [BB05]. The domains range from social to natural and engineering sciences, and lead to many dimensions. Without going into detail, we alter and adjust the concluding remarks from [BB05] and define:

> The context in modeling comprises static environmental data that might influence system behavior in producing or reutilizing model recommendations.

The static in this definition implies that we take a moment in time of a modeling activity and capture the present data. This data is environmental because it gathers the current state of tools and the recent actions which led to it. However, the impact of these data on system behavior is not guaranteed, because each individual model recommendation candidate generation (ϱ_{gen}) determines whether the data are used. Finally, the produced model recommendations must suit the context and, because environmental data comprise tool information, this means that a context enables reutilizations. Concretely, a model given in operation-based format is transformed so that it can be applied to or inserted into the tool. In a sense, we can subdivide contexts into querying and inserting contexts. The former are as mentioned above, whereas the latter are transformation engines. This makes our context universe ($\mathcal{C}$):

$$\mathcal{C} := \text{TOOL} \times \text{VIEW} \times \tau \times \text{m} \times \text{MPROPERTIES} \times \text{EDITING} \times ... \tag{3.78}$$

$$\tau \in \{\text{reactive}, \text{proactive}\} \tag{3.79}$$

$$\text{m} := (\text{E}_\text{A}, \text{E}_{cl}, \text{E}_\text{C}, ..., \rho_{\text{eAttributes}}, \rho_{\text{eClassifiers}}, ..., \mathfrak{R}) \in \mathcal{M} \tag{3.80}$$

$$\text{MPROPERTIES} := \{(\text{dit}, v), ... (\text{TERM}_i, \text{<itemname>}), ...\}, \text{TERM}_i \in \text{STR}, v, i \in \mathbb{N} \tag{3.81}$$

$$\text{EDITING} := \{\mathfrak{Z}^{\texttt{all}}, \mathfrak{Z}^{\texttt{cre}}, \mathfrak{Z}^{\texttt{del}}, \mathfrak{Z}^{\texttt{sel}}\}, \mathfrak{Z}^{\texttt{all}}, \mathfrak{Z}^{\texttt{cre}}, \mathfrak{Z}^{\texttt{del}}, \mathfrak{Z}^{\texttt{sel}} \in \mathcal{M}^{\mathfrak{Z}} \tag{3.82}$$

$$\text{TERMS} := \bigtimes(\text{TERM}_i, \text{val}_i), \qquad \text{TERM}_i \in \text{STR}, \text{val}_i, i \in \mathbb{N} \tag{3.83}$$

Note that we consider this context data to be extendable, as indicated by the context components in figure 3.33, and that some data in equation (3.78) depend on implementation details. For example, a TOOL in equation (3.78) is derived at runtime as some identifier. For our realization, EMF Ecore editors are supported by individual contexts, because a context component in figure 3.33 can deliver a context $\text{c} \in \mathcal{C}$. Further, many modeling tools gather content information in one place and allow several perspectives, which are often called views. These views are represented by an identifier denoted as VIEW. This is also derived at runtime. We do not go into detail regarding how the actual data are monitored or captured, and consider it given. Instead, we focus on conceptually relevant parts, such as the trigger method (τ) in equation (3.79). This represents how a context is evaluated, i.e., an explicit trigger of a recommender strategy leads to a reactive context, whereas an implicit trigger, e.g., timeout or select, is represented by a proactive context. Other than that, the model ($\text{m} \in \mathcal{M}$ cf. equation (3.1) (p. 36)) is the model in tuple form that is currently being edited. Subsequently, we go into further detail regarding the MPROPERTIES, EDITING, and TERMS of our universe of contexts ($\mathcal{C}$).

MPROPERTIES characterize the features of models (cf. equation (3.30) (p. 60)), though recommender strategies could conduct the same calculations. In detail, the properties provided are mostly related to model metrics and graph clustering, as explained in subsection 3.3.2 (p. 75). For example, associate and god-class clustering provide potentially essential features for further processing, and we put them in MPROPERTIES as follows: (TERM, <itemname>). Note that MPROPERTIES are not meant for optimization, because extracting the computationally expensive Girvan–Newman or Kernighan–Lin clustering (cf. subsection 3.3.2 (p. 75)) from recommender strategies would break the separation of concerns.

The EDITING, which we declared in equation (3.82), comprises sequences in operation-based format, as explained in equation (3.16) (p. 39), i.e., the sequences are $\in \mathcal{M}^{\mathfrak{I}}$ but contain some unique properties. As the names indicate, each sequence only contains edits related to its name, e.g., create operations (π_c cf. equation (3.5) (p. 38)). However, persisting a sequence of redundant operations, which have already been "applied" to a model, is of little help in our case, so the monitoring stores find operations (φ) instead. Note that this also solves the question of how to represent select operations. Further, it is a non-intrusive representation, so a recommender strategy can immediately use this for analysis, given the model (m).

Although we do not go into detail about monitoring operations, note that operations generally cannot be gleaned from tools easily, and we approach this by monitoring and then investigating raw interactions in tiers, as done for raw sensor data in context management [Fra01; Bet+10]. "Tier 0" is the level of sensors that capture actions as clicks. On "Tier 1", the raw data that have been captured are described. In our case, this can be an interaction for creating a class ($\pi_c(\mathrm{E}_{\mathrm{C}}^{\mathtt{name}})$). These raw data differ with the implementation, because editors often create prototype objects [Gam+95] that are adjusted in successive steps. Hence, "Tier 2" filters such noise, so that "Tier 3" can alter them to φ operations and assign them to the right sequence. Hence, in our example, $\pi_c(\mathrm{E}_{\mathrm{C}}^{\mathtt{name}})$ becomes $\varphi_{\mathrm{E_C}}^{\mathrm{NAME}}$ in $\mathfrak{I}^{\mathtt{cre}}$. Finally, "Tier 4" adds semantic meaning, which we postpone and leave to the recommender strategies. For now, consider a mouse release with several selections. A recommender strategy can take this as a starting point for a reactive `isomorphic Query` with the selected elements.

The TERMS introduced in equation (3.83) are the result of additional environmental analysis. Indirectly related files are taken into account and prepared for further processing. For example, requirements documents, glossaries, or other project files can provide grounds for term analysis and determine potentially valuable information. Tag clouds are prominent examples, and we provide a tuple form of terms comprising the term and a respective inverse document frequency as a value [LGS11]: (*term*, <value>).

As a marginal note, we should contrast our context universe ($\mathcal{C}$) with "interaction data" defined as "a record of actions taken by a user with a tool" [MFR14]. In our case, the defined "types of data" are as follows: "actions" are EDITING sequences ($\mathfrak{I}^{\mathtt{all}}$, ...), the "artifacts" are models (m), and the "tools" are a single TOOL, i.e., editor. For the "type of data" denoted as "context" [MFR14], which is seen as an active issue or task, we introduce terms, but could consider Mylyn tasks as well. For now, this is out of scope.

With our universe of contexts ($\mathcal{C}$) and our context analysis operation (ϱ_{ana}) declared in equation (3.72), we can explain how this operation works. In terms of context-aware recommender systems, it is a contextual-prefiltering [AT11]. Hence, it alters (α_{ALT}) the monitored context (c) for a given QUERYTERM, and adjusts (α_{ADJ}) the latter as shown in equation (3.84), regardless of stemming and stop word removal [Por80; Wil06; MRS08]. In the case of proactive triggering (cf. equation (3.86)), the most recent operation is used to initialize a QUERYTERM. We distinguish three cases: First, a create operation leads to a regular QUERYTERM built by the given name. Second, a select operation results in an isomorphic QUERYTERM by means of the name and related selections. Third, a delete operation induces a neighbor QUERYTERM by name (intra- or inter-model). Equation (3.87) exemplifies this in a semi-formal notation. It looks at the last operation added ($\mathfrak{Z}^{\texttt{all}}$) and determines which editing sequence it belongs to, then builds the corresponding add-on denoted Q'. If the last operation was a create operation, Q' becomes ":wname '<name>'" or "$\Psi(\mathfrak{l}_{\text{WName}})$, <name>" and uses the index for "words in names" ($\mathfrak{l}_{\text{WName}}$), as introduced in table 3.7 (p. 57). We leverage editing sequences from a given context (c) and discuss this in more detail, especially how to build Q', in subsection 3.5.7.

$$\varrho_{ana} := \mathcal{KL} \times \mathcal{C} \times \text{STR} \rightarrow \mathcal{KL} \times \mathcal{C} \times \text{STR} \times \mathcal{MRC} :$$
$$(\mathbb{KL}, \text{c}, \text{QUERYTERM}) \mapsto (\mathbb{KL}, \alpha_{\text{ALT}}(\text{c}), \alpha_{\text{ADJ}}(\text{c}, \text{QUERYTERM}), \{(\varnothing, ...)\}) \tag{3.84}$$

$$\alpha_{\text{ALT}} := \mathcal{C} \rightarrow \mathcal{C} :$$
$$(\text{c}) \mapsto \text{c}', \quad \text{with MPROPERTIES as in equation (3.81)} \tag{3.85}$$

$$\alpha_{\text{ADJ}} := \mathcal{C} \times \text{STR} \rightarrow \text{STR} :$$
$$(\text{c}, \text{QUERYTERM}) \mapsto \text{QT'}, \text{QT'} := \text{QUERYTERM} + \begin{cases} \text{""}, & \tau = \text{reactive} \\ \text{Q'}, & \tau = \text{proactive} \end{cases} \tag{3.86}$$

$$\mathfrak{Z}^{\texttt{all}} = \bigcirc_{i=0}^{n \in \mathbb{N}} \pi_i \rightarrow \pi_n = \begin{cases} \pi_c \leftarrow \bigcirc_{i=0}^{c \in \mathbb{N}} \pi_i = \mathfrak{Z}^{\texttt{cre}} & \Rightarrow \text{Q'} := \Psi(\mathfrak{l}_{\text{WName}}), \text{“name”} \\ \pi_s \leftarrow \bigcirc_{i=0}^{s \in \mathbb{N}} \pi_i = \mathfrak{Z}^{\texttt{sel}} & \Rightarrow \text{Q'} := \text{:iso “name”} \\ \pi_d \leftarrow \bigcirc_{i=0}^{d \in \mathbb{N}} \pi_i = \mathfrak{Z}^{\texttt{del}} & \Rightarrow \text{Q'} := \neg(\Psi(\mathfrak{l}_{\text{WName}}), \text{“name”}) \end{cases} \tag{3.87}$$

Let us now skip forward to the moment after an entire model recommendation production has been completed, e.g., $\varrho_{fil} \circ \varrho_{rnk} \circ \varrho_{gen} \circ \varrho_{ana}$, and a recommendation is picked for reutilization. This might require a final contextual-postfiltering [AT11], because our model (m) is then provided in operation-based format ($\mathfrak{Z}$) for the purpose of platform independence. However, this means that the operations need to be transformed to the given target platform, i.e., TOOL, so that the model can be reutilized regardless of whether a textual, tree, or graphical editor is the application target. In addition, a VIEW of a model can differ from its content. Hence, elements that were taken into account during the recommendation process might not be contained in a VIEW but are necessary for reutilizations. The context components in figure 3.33 take care of both. We go into some detail about this when discussing our realization in section 4.5 (p. 156).

3.5.5. Recommendation Candidate Generation

Given a knowledge library, we study the field of knowledge-based recommender systems, and extend this to contextual knowledge-based recommender systems as soon as we leverage a context. This subdivides model recommendation candidate generation (ϱ_{gen} cf. equation (3.73)) depending on whether we use one or the other, so we need to provide operations for both. This forms the basis for a complete model recommendation generation algorithm (ϱ_{gen}) matching our domain and fostering a model recommendation candidates universe ($\mathcal{MRC}$) [BR11]. Hence, we first describe the investigation of basic knowledge-base results called model recommendation candidates. These are results from our knowledge library without using its structural form, but based on the `Query` capabilities by means of a given context. Second, we further process these model recommendation candidates by employing the structural features of our knowledge library for the granularity and relatedness properties of model recommendation candidates.

A starting point for the model recommendation generation algorithm or its results is a lightweight requirements specification, which we call purpose. This is meant to grasp the intention of a model recommendation as model `purposes` for model evolution in subsection 3.4.4 (p. 101). However, this time, the purpose does not rely on the quality model, but on the model recommendation properties introduced in table 3.10. As the extent is subdivided into granularity and relatedness, and because impact is irrelevant in our case, we can denote a purpose as a triple containing the QUERYTERM, leveraged context, and the range regarding extent. In other words, the QUERYTERM is accompanied by statements of sensitivity and extent $(\text{QUERYTERM}, \mathsf{c}, \{\text{id}, \text{id}, ...\})$.

For example, $(\text{AIRPORT}, \mathsf{c}, \{\text{element, complete, chain no crosslink}\})$ with a considered context $\mathsf{c} = \{\text{“”}, \text{“”}, \text{“”}, \mathsf{m}, \{\varnothing, ...\}, \text{TERMS}\}$ relies on a QUERYTERM for `Airport`, a given model (m), and some TERMS. It generates model recommendations candidates of size element, complete, and chain without cross-links. Later, we will denote the sizes with identifiers, e.g., EL, C, CH^{-}. Other contextual information that is possibly provided but not used is, e.g., the EDITING or MPROPERTIES. In subsection 3.5.6, we will discuss possible purposes in more detail and describe the impact of contexts. For example, the trigger (τ) impacts the extent because proactive model recommendation should provide smaller recommendations.

An addition to that starting point is a universe of model recommendation candidates ($\mathcal{MRC}$). The tracking of candidates is supported in rounds of generation. The layout is in line with granularity and relatedness, as introduced in table 3.11. Each comprised set is meant for a specific tracking purpose (cf. identifiers in equation (3.88)), so candidate origins can support ranking and filtering in subsection 3.5.6. In detail, each set contains tuples as shown in equation (3.89). The last two elements contain the `Model` and its respective QUERYTERM; these lead to the candidate, which is the first element.

$$\mathcal{MRC} := \bigtimes \mathcal{MRC}_{\text{ID}} \tag{3.88}$$

$$\mathcal{MRC}_{\text{ID}} := \bigtimes (\mathsf{E}_{\mathsf{M}}, \mathsf{E}_{\mathsf{M}}, \text{STR}), \text{ID} \in \text{CAND-IDS} \tag{3.89}$$

$$\text{CAND-IDS} := \{\text{EL}, \text{S}, \text{C}, \text{CH}^{+}, \text{CH}^{-}, \text{G}, \text{CAT}\} \tag{3.90}$$

Sensitivity: For a given knowledge library ($\mathbb{KL} \in \mathcal{KL}$), consider the search for model recommendation candidates for a QUERYTERM "$\Psi(\mathrm{I}_{\mathrm{WModel}})$, NAME" ($\Phi_{\mathbb{KL}}^{\mathrm{QUERYTERM}}$). This is probably the simplest and most intuitive way to use our knowledge library, and it will provide a set of `Models` matching "NAME". However, we should leverage rather more of the index and `Query` functionality available (see subsection 3.2.2 (p. 56) and subsection 3.2.3 (p. 61)), given that we have more information at hand, i.e., contextual information ($\mathrm{c} \in \mathcal{C}$). Thus, we look into model recommendation candidate generation (ϱ_{gen}) and refine equation (3.73) as follows:

$$\varrho_{gen}^{x-\mathrm{SENS}} := \mathcal{KL} \times \mathcal{C} \times \mathrm{STR} \times \mathcal{MRC} \rightarrow \mathcal{KL} \times \mathcal{C} \times \mathrm{STR} \times \mathcal{MRC} : \tag{3.91}$$

$$(\mathbb{KL}, \mathrm{c}, \mathrm{QUERYTERM}, \mathrm{mrc}) \mapsto (\mathbb{KL}, \mathrm{c}, \mathrm{QUERYTERM}, \mathrm{mrc'}), \tag{3.92}$$

$$\forall \varepsilon_{\mathrm{M}} \in \Phi_{\mathbb{KL}}^{\mathrm{QUERYTERM}} \Rightarrow \mathrm{mrc'_c} := \mathrm{mrc_c} \cup \{(\varnothing, \varepsilon_{\mathrm{M}}, \mathrm{QUERYTERM})\}, \mathrm{mrc'_c} \in \mathrm{mrc'}$$

The "$x-$ SENS" for x-"sensitivity" in this operation already indicates that we mean to summarize several generation operations. The results only alter the set of model recommendation candidates (mrc') according to the requested sensitivity, and do so by using a different QUERYTERM that largely relies on one index. We could employ the other indexes introduced for our knowledge library or graph walk strategies, and this will be part of the discussion in subsection 3.5.7. For example, we will then think of fallback strategies in case the number of model recommendation candidates is too low. For now, we approach candidate generation from the perspective of QUERYTERMS and contextual information, and use the abovementioned x in equation (3.92) as a link between the name of the generation operation and its respective QUERYTERM in equations (3.93) to (3.98). For example, a BASIC-SENSITIVITY generation operation uses the QUERYTERM given in equation (3.93).

$$\varrho_{gen}^{\text{BASIC-SENS}}: \quad \mathrm{QUERYTERM} := (\text{:wmodel "queryterm"}) \tag{3.93}$$

$$\varrho_{gen}^{\text{SYNO-SENS}}: \quad \mathrm{QUERYTERM} := (\mathrm{OR}\ (\text{:wmodel "synonym-1"})\ ...) \tag{3.94}$$

$$\varrho_{gen}^{\text{PROP-SENS}}: \quad \mathrm{QUERYTERM} := (\mathrm{OR}\ (\text{:wmodel } \mathrm{TERM}_1)\ ...), \tag{3.95}$$

$$(\mathrm{TERM}_i, \text{<itemname>}) \in \mathrm{MPROPERTIES} : i \in \mathbb{N}$$

$$\varrho_{gen}^{\text{TERMS-SENS}}: \quad \mathrm{QUERYTERM} := ((\text{:wmodel } \mathrm{TERM}_j)\ ...), \tag{3.96}$$

$$(\mathrm{TERM}_j, \mathrm{val}_j) \in \mathrm{TERMS} : \forall k \in \mathbb{N} \Rightarrow \mathrm{val}_k \leqslant \mathrm{val}_j$$

$$\varrho_{gen}^{\text{ISO-SENS}}: \quad \mathrm{QUERYTERM} := (\text{:iso } (\mathrm{OR}\ (\text{:wmodel } \texttt{name-1}\ )\ ...), \tag{3.97}$$

$$\varepsilon_{\mathrm{C}}^{\texttt{name-i}} \in \mathrm{E_C}, \mathtt{i} = 1, ...|\mathrm{E_C}|, \mathrm{E_C} \in \mathrm{m}$$

$$\varrho_{gen}^{\text{EDIT-SENS}}: \quad \mathrm{QUERYTERM} := (\text{:iso } (\mathrm{OR}\ (\text{:wmodel } \texttt{name-1}\ )\ ...), \tag{3.98}$$

$$\texttt{name-i} \in \mathrm{STR}, \varphi_{\gamma}^{\texttt{name-i}} = \pi_i \leftarrow \bigcirc_{j=0}^{s \in \mathbb{N}} \pi_j \text{ of } \mathfrak{Z}^{\texttt{sel}},$$

$$\gamma = \mathrm{E_C}, 0 \leqslant \mathrm{s} \leqslant |\ \mathfrak{Z}^{\texttt{sel}}\ |$$

$$\varrho_{gen}^{\text{PROP-TERMS-SENS}}: \quad \mathrm{QUERYTERM} := (\mathrm{OR}(\text{ equation (3.95)}\ )\ (\text{ equation (3.96)}\ ) \tag{3.99}$$

The most basic model recommendation candidate generation operation, which we denote BASIC-SENS, is introduced by means of its "queryterm" in equation (3.93). It uses the "words in model" index and queries the knowledge library, i.e., builds a QUERYTERM from a given input "queryterm" as follows: ":wmodel 'queryterm'". Hence, it results in candidates with the query term in nothing but the model. This means that no meta-information is considered. A slightly altered version of the BASIC-SENS generation QUERYTERM is also prefixed by ":wmodel", but changes the "queryterm" and substitutes it with synonyms. We do not go into detail about synonyms here, but note that this often leads to multiple rounds of candidate generation, i.e., `Query`s, or a disjunctive QUERYTERM, as shown in equation (3.94). Note that these two generation operations rely on given query terms, which are particularly suited to reactive model recommendation production.

In contrast, and better-suited to proactive model recommendation production, the PROP-SENS operation is introduced by means of its QUERYTERM in equation (3.95), using TERMS derived from MPROPERTIES. These are class names that are considered "central" and are expected to be supportive for model recommendation candidate generation. Hence, the QUERYTERM is disjunctive and built by these terms employing the ":wmodel" index. An alternative to this generation operation is the TERMS-SENS operation, which we introduced by means of a QUERYTERM in equation (3.96). This uses the TERMS provided by a context, taking the most highly ranked TERM and building a QUERYTERM from it using the ":wmodel" index. Hence, it looks at models of `Model`s only and ignores meta-data. Note that both of these QUERYTERMS depend on the number of TERMS employed, and we build an example for the latter using only one term. A realization might consider several directions, as we discuss in subsection 3.5.7. For now, note that the disjunctive nature of these QUERYTERMS can lead to many model recommendation candidates.

Another model recommendation candidate generation operation, which is mostly suited to proactive use, approaches the structural comparison of model graph structures, i.e., isomorphism. This operation builds on a QUERYTERM comprising the names of the classes in the currently edited model, as shown in equation (3.97). It is prefixed by ":iso", which is not an index but a `Query` leveraging several indexes as necessary. In fact, a realization (i.e., `Query`) building on this type of QUERYTERM takes a model as a parameter: recall that an `IsomorphicQuery` on a knowledge library for an ISO-SENS generation operation involves more processing than the other QUERYTERMS we have introduced so far. As we explained in subsection 3.2.3 (p. 61), the structural information, i.e., `IsomorphicQuery`, works in two rounds. In the first round, isomorphic candidates are queried using the class names. Structural information is then leveraged on these isomorphic candidates to find actual isomorphic (sub-)models. This is possible, because structural information is available in our context through the currently edited model (m). Similarly, the QUERYTERM built in equation (3.98) leverages the ":iso" index, but builds it on the elements recently selected from the model that is currently being edited. These can be extracted from the find operations in the select sequence ($\Im^{\mathtt{sel}}$). Note that the upper bound remains to be set.

Finally, we combined two generation operations in a QUERYTERM for a PROP-TERMS-SENS operation. The disjunction in equation (3.99) indicates the substitution of two successive

operations, i.e., $\varrho_{gen}^{\text{PROP-SENS}} \circ \varrho_{gen}^{\text{TERMS-SENS}}$. This is an example of combining generation operations, which we will do with a "sensitivity extension schema" in subsection 3.5.7.

Granularity and Relatedness: Consider the model recommendation candidates (mrc $\in \mathcal{MRC}$) gained by means of a QUERYTERM "$\Psi(\text{I}_{\text{WModel}})$, NAME" ($\Phi_{\mathbb{KL}}^{\text{QUERYTERM}}$), i.e., $\varrho_{gen}^{\text{BASIC-SENS}}$, given a knowledge library ($\mathbb{KL} \in \mathcal{KL}$). This makes all the candidates in mrc, i.e., "complete candidates" (mrc_C), so the `Models` represent entire models. At times, we seek a different size of model recommendation candidates, as introduced in table 3.11. We can derive these new model recommendation candidates of different granularity or relatedness for each model recommendation candidate ε_M (cf. table 3.5 (p. 55)) semi-formally:

$$\begin{aligned}\varrho_{gen}^{\text{ID}} :=& \mathcal{KL} \times \mathcal{C} \times \text{STR} \times \mathcal{MRC} \to \mathcal{KL} \times \mathcal{C} \times \text{STR} \times \mathcal{MRC} : \\ & (\mathbb{KL}, \text{c}, \text{QUERYTERM}, \text{mrc}) \mapsto (\mathbb{KL}, \text{c}, \text{QUERYTERM}, \text{mrc'}), \\ & \text{mrc'}:=(\,...\,\text{mrc'}_\text{C}\,...\,), \forall(\varnothing, \varepsilon_\text{M}, \text{QUERYTERM}) \in \text{mrc}_\text{C} \\ & \qquad \Rightarrow \text{mrc'}_\text{ID} := \text{mrc}_\text{ID} \cup \text{mrc'}_\text{ID} \cup \text{set}_\text{ID},\ \text{ID} \in \text{CAND-IDS} \\ & \qquad \text{and } \text{set}_\text{ID} \text{ as in equations (3.102) to (3.107) and (3.109)}\end{aligned} \tag{3.100}$$

This rather abstract operation delivers a set of tuples in a model recommendation candidates set (mrc') containing the result as a triple, as we explain shortly. Further, we keep track of the origin of each model recommendation candidate (ε_M) with a set identifier (ID), and even forward the origin as its respective QUERYTERM. An example of an identifier is EL for element candidates. These types of candidates can be of single size only, as attributes or classes (E_A or E_C). Other candidates can be larger, e.g., G for sets of group-related candidates. This additional information will aid the ranking of model recommendation candidates in subsection 3.5.6. In detail, equation (3.100) denotes several operations, e.g., for $\varrho_{gen}^{\text{EL}}$ or $\varrho_{gen}^{\text{CL}^+}$, which use the following sets for fixed ε_M:

$$\begin{aligned}\text{With m} &:= \varepsilon_\text{M}.\text{files, as } (\text{E}_\text{A}, \text{E}_{cl}, \text{E}_\text{C}, ..., \rho_{\text{eAttributes}}, \rho_{\text{eClassifiers}}, ..., \mathfrak{R}) \\ \text{element: } \text{set}_\text{EL} &:= \text{see equation (3.109)} && (3.101) \\ \text{submodel: } \text{set}_\text{S} &:= \bigcup (\varepsilon_i, \varepsilon_\text{M}, \text{QUERYTERM}), && (3.102) \\ & \quad \text{s}_i := (\text{E}_\text{A}|_\text{S}, \text{E}_{cl}|_\text{S}, \text{E}_\text{C}|_\text{S}, ..., \rho_{\text{eAttributes}}|_\text{S}, \rho_{\text{eClassifiers}}|_\text{S}, ..., \mathfrak{R}), \\ & \quad \varepsilon_i.\text{files}:=\text{s}_i \subseteq \text{m, by restriction cf. subsection 3.3.2 (p. 75)} \\ \text{complete: } \text{set}_\text{C} &:= \{(\varepsilon_\text{M}, \varepsilon_\text{M}, \text{QUERYTERM})\}, && (3.103) \\ \text{chain: } \text{set}_{\text{CH}^+} &:= \bigcup (\varepsilon_i, \varepsilon_\text{M}, \text{QUERYTERM}), && (3.104) \\ & \quad \varepsilon_i \in \{\varepsilon \in \text{E}_\text{M} : \exists \varepsilon_\text{C} \in \text{E}_\text{C},\ \varepsilon_\text{C}.\text{syntactics} \neq \text{""},\ (\varepsilon, \varepsilon_\text{C}), (\varepsilon_\text{C}, \varepsilon_\text{M}) \in \mathbb{KL}_\text{E}\} \\ \text{chain: } \text{set}_{\text{CH}^-} &:= \bigcup (\varepsilon_i, \varepsilon_\text{M}, \text{QUERYTERM}), && (3.105) \\ & \quad \varepsilon_i \in \{\varepsilon \in \text{E}_\text{M} : \exists \varepsilon_\text{C} \in \text{E}_\text{C},\ \varepsilon_\text{C}.\text{syntactics} = \text{""},\ (\varepsilon, \varepsilon_\text{C}), (\varepsilon_\text{C}, \varepsilon_\text{M}) \in \mathbb{KL}_\text{E}\} \\ \text{grouped: } \text{set}_\text{G} &:= \bigcup (\varepsilon_i, \varepsilon_\text{M}, \text{QUERYTERM}), && (3.106) \\ & \quad \varepsilon_i \in \{\varepsilon \in \text{E}_\text{M} : \exists \varepsilon_\text{G} \in \text{E}_\text{G},\ (\varepsilon, \varepsilon_\text{G}), (\varepsilon_\text{G}, \varepsilon_\text{M}) \in \mathbb{KL}_\text{E}\} \\ \text{categorized: } \text{set}_\text{CAT} &:= \bigcup (\varepsilon_i, \varepsilon_\text{M}, \text{QUERYTERM}), && (3.107) \\ & \quad \varepsilon_i \in \{\varepsilon \in \text{E}_\text{M} : \exists \varepsilon_\text{Cat} \in \text{E}_\text{Cat},\ (\varepsilon, \varepsilon_\text{Cat}), (\varepsilon_\text{Cat}, \varepsilon_\text{M}) \in \mathbb{KL}_\text{E}\}\end{aligned}$$

The above equations make use of several semi-formal notations. First, we use the dot notation to access parts of `LibraryElements`, e.g., $\varepsilon_{\mathbb{M}}$.files, which is actually a `URI`, to get the underlying model in tuple notation or check whether a cross-link is available, i.e., syntactic information in a `Connector` ($\varepsilon_{\mathbb{C}}$.syntactics$\neq$"").

Furthermore, most of the sets from equations (3.102) to (3.107) and (3.109) determine related parts through an intermediate element. For example, grouped `Models` need a `Group` in between, so this `Group` must exist and relate to another `Model`, as in equation (3.106). Though this is similar for `Categorys` and `Connectors`, i.e., cross-links, the intermediate elements differ for model internal model recommendation candidate generation. For submodel model recommendation candidates, this applies the algorithm in terms of restricting functions and clustering, as we defined in subsection 3.3.2 (p. 75). For elements, the details are given below.

We outsourced equation (3.101) to equation (3.109) to highlight its specialty. There is one additional parameter that must be taken into account when generating these recommendations: γ. We can obtain this parameter from a QUERYTERM or determine it in a proactive triggering from our context EDITING sequence, as in equation (3.108). In the latter case, we can get the necessary type information by means of the last operation applied. This is contained in $\mathfrak{Z}^{\mathtt{all}}$ as a find operation (φ), which provides the last-used element, and we can use our type determining operation (Γ). Let an attribute (E_{A}) be the last element created. Then, the element recommendation should be of the same type. Note that this does not forbid model recommendation candidates of other granularity from being generated at the same time. The former case regarding the additional parameter with QUERYTERM is only mentioned for the sake of completeness, because its application is rather theoretical. This is because of the proactive nature of this size of model recommendation candidates. In case an attribute has just been created, it makes sense to proceed as explained above, but in the case of reactive model recommendation production, the quicker way is simply to create the intended element. Bearing in mind that our EDITING from equation (3.82) keeps finding operations for the respective models, we can determine γ and generate model recommendation candidates of element granularity (cf. table 3.1 (p. 34) and table 3.2 (p. 36)):

$$\text{with } \gamma \quad := \begin{cases} \Gamma(\pi_n), \pi_n \leftarrow \bigcirc_{i=0}^{n\in\mathbb{N}} \pi_i = \mathfrak{Z}^{\mathtt{all}}, & \tau = \text{proactive} \\ \text{given by QUERYTERM}, & \tau = \text{reactive} \end{cases} \tag{3.108}$$

$$\text{and } \varepsilon_\gamma \quad := \pi_n, \leftarrow \bigcirc_{i=0}^{n\in\mathbb{N}} \pi_i = \mathfrak{Z}^{\mathtt{all}} \text{ we can get:}$$

$$\begin{aligned} \text{element: set}_{\mathsf{EL}} \quad := \; & \bigcup (\varepsilon_i, \varepsilon_{\mathbb{M}}, \text{QUERYTERM}), \\ & \forall \varepsilon_j \in \{\varepsilon \in \mathsf{E}_\gamma \,:\, \exists \varepsilon_{ne} \in \mathsf{E}_{ne},\, (\varepsilon, \varepsilon_{ne}), (\varepsilon_{ne}, \varepsilon_\gamma) \in \rho_{\mathsf{e}}\} \\ & \quad \cup \{\varepsilon \in \mathsf{E}_{\mathsf{C}} \,:\, (\varepsilon, \varepsilon_\gamma) \in \rho_{\mathsf{eSuperTypes}}\} \\ & \Rightarrow \mathsf{e}_j := (\mathsf{E}_{\mathsf{A}}|_{\mathsf{e}}, \mathsf{E}_{cl}|_{\mathsf{e}}, \mathsf{E}_{\mathsf{C}}|_{\mathsf{e}}, \dots, \rho_{\mathsf{eAttributes}}|_{\mathsf{e}}, \rho_{\mathsf{eClassifiers}}|_{\mathsf{e}}, \dots, \mathfrak{N}), \\ & \quad \mathsf{E}_\gamma|_{\mathsf{e}} = \{\varepsilon_j\}, \varepsilon_i.\text{files:=}\mathsf{e}_j \end{aligned} \tag{3.109}$$

Once again, the sought elements are related to the query, so the intermediate elements

need to be found. We do so by allowing E_{ne} and, in the case where Γ is E_A, the generated elements are as well, because the container is a class that is an E_{ne}. Note that this also works for E_C, because E_R "bridge", but superclasses are filtered out. Hence, E_C directly related by $\rho_{\text{eSuperTypes}}$ are allowed, because only elements of E_C have such relations.

Altogether, the number of model recommendation candidates that can be generated by means of our ϱ_{gen} operations can become quite large. Hence, reasonable ranking and filtering is necessary, and we introduce these operations in subsection 3.5.6.

Example: Given our knowledge library ($\mathbb{KL}^{\textit{figure 3.7}}$) from equation (3.38) (p. 61), as depicted in figure 3.7 (p. 53), we can illustrate ϱ_{gen} as shown in table 3.12. For $\varrho^{\text{EL}}_{gen}$, an `Airport` class is used as input and results in all classes except $\varepsilon_{\text{C}}^{\texttt{Airport}}$. Note that this class is part of the tuple anyway, because it was part of the QUERYTERM. Further, the tuple contains the `Model` ($\varepsilon_{\text{M}}^{\texttt{Airport}}$), because this is the source of the respective elements. The other exemplary QUERYTERMs use an index and `Airport`. Hence, `Airport` is part of the tuples, as is the $\varepsilon_{\text{M}}^{\texttt{Airport}}$. The latter is of particular importance for cross-links, i.e., $\varrho^{\text{CH}^+}_{gen}$, because it eases processing. Finally, note that $\varrho^{\text{CH}^-}_{gen}$ leads to an empty set because `syntactic` information is provided in our example. This means that the $\varepsilon_{\text{M}}^{\texttt{Airport}}$ has no `Connectors` without `syntactic` information.

Table 3.12.: Examples for ϱ_{gen} operation as in equations (3.102) to (3.107) and (3.109) for figure 3.7 (p. 53)

Operation	QUERYTERM	Result
$\varrho^{\text{EL}}_{gen}$	$\varepsilon_{\text{C}}^{\texttt{Airport}}$	$\{(\varepsilon_{\text{C}}^{\texttt{Checkin}}, \varepsilon_{\text{M}}^{\texttt{Airport}}, \varepsilon_{\text{C}}^{\texttt{Airport}}), \ldots$ $(\varepsilon_{\text{C}}^{\texttt{Tower}}, \varepsilon_{\text{M}}^{\texttt{Airport}}, \varepsilon_{\text{C}}^{\texttt{Airport}})\}$
ϱ^{S}_{gen}	$\Psi(\mid_{\text{Name}})$, `Airport`	see figure 2.3 (p. 27) and subsection 3.3.2 (p. 75)
ϱ^{C}_{gen}	$\Psi(\mid_{\text{Name}})$, `Airport`	$\{(\varepsilon_{\text{M}}^{\texttt{Airport}}, \varepsilon_{\text{M}}^{\texttt{Airport}}, \texttt{Airport})\}$
$\varrho^{\text{CH}^+}_{gen}$	$\Psi(\mid_{\text{Name}})$, `Airport`	$\{(\varepsilon_{\text{M}}^{\texttt{Passenger}}, \varepsilon_{\text{M}}^{\texttt{Airport}}, \texttt{Airport}),$ $(\varepsilon_{\text{M}}^{\texttt{Vehicle}}, \varepsilon_{\text{M}}^{\texttt{Airport}}, \texttt{Airport})\}$
$\varrho^{\text{CH}^+}_{gen}$	$\Psi(\mid_{\text{Name}})$, `Airport`	$\{\varnothing\}$
ϱ^{G}_{gen}	$\Psi(\mid_{\text{Name}})$, `Airport`	$\{(\varepsilon_{\text{M}}^{\texttt{Passenger}}, \varepsilon_{\text{M}}^{\texttt{Airport}}, \texttt{Airport}),$ $(\varepsilon_{\text{M}}^{\texttt{Vehicle}}, \varepsilon_{\text{M}}^{\texttt{Airport}}, \texttt{Airport})\}$
$\varrho^{\text{CAT}}_{gen}$	$\Psi(\mid_{\text{Name}})$, `Airport`	$\{(\varepsilon_{\text{M}}^{\texttt{Station}}, \varepsilon_{\text{M}}^{\texttt{Airport}}, \texttt{Airport}),$ $(\varepsilon_{\text{M}}^{\texttt{Harbor}}, \varepsilon_{\text{M}}^{\texttt{Airport}}, \texttt{Airport})\}$

3.5.6. Recommending the Appropriate

Regular content recommender systems often employ heuristics or metrics such as term frequency-inverse document frequency (tf-idf) to assess candidates [Men14]. These approaches often rely on term redundancy, which is not feasible in our case because redundancy rarely happens in models. Hence, any ranking for our approach will need to

use contextual information as well as the QUERYTERM to assess, and eventually rank, model recommendation candidates so they can become model recommendations.

This ranking narrows down each model recommendation candidate to a value, which means that we transform candidate tuples of the form (E_M, E_M, STR) to actual model recommendations by assigning them with a real number. Thereafter, we add a ranking value to this tuple and call it a model recommendation. These all belong to the model recommendations universe ($\mathcal{MR}$), which is similar to the model recommendations candidates universe from equation (3.89) and uses the known candidate identifiers:

$$\mathcal{MR} := \bigtimes \mathcal{MR}_{\text{ID}} \tag{3.110}$$

$$\mathcal{MR}_{\text{ID}} := \bigtimes (E_M, \text{rnk}, E_M, \text{STR}), \text{ID} \in \text{REC-IDS}, \text{rnk} \in \mathbb{R}^+ \tag{3.111}$$

$$\text{REC-IDS} := \text{CAND-IDS} \text{ cf. equation (3.90)} \tag{3.112}$$

Bear in mind that we retain the structure, i.e., every model recommendation in its granularity and sensitivity identifying set. This will simplify the ranking and filtering, especially in contextual-postfiltering, as we explain later. Additionally, we rename the set of candidate identifiers for comprehensibility.

Ranking: The ranking (ϱ_{rnk} in equation (3.74)) that we introduce works in steps and has a more algorithmic nature than the previous explanations. Therefore, we eventually digress from our previous format of illustration and provide pseudocode 3.6. Still, we build up to this overview through a given environment: for ranking, we can leverage a given QUERYTERM, context c ($\in \mathcal{C}$), and set of model recommendation candidates mrc ($\in \mathcal{MRC}$) to gain model recommendations $\text{mr} = (\text{mr}_{\text{EL}}, ...)$ ($\in \mathcal{MR}$):

$$\varrho_{rnk} := \mathcal{KL} \times \mathcal{C} \times \text{STR} \times \mathcal{MRC} \to \mathcal{C} \times \mathcal{MR} :$$

$$\mathbb{KL}, \text{c}, \text{QUERYTERM}, (\text{mrc}_{\text{EL}}, \text{mrc}_{\text{S}}, \text{mrc}_{\text{C}}, ...) \mapsto \text{c}, (\text{mr}_{\text{EL}}, \text{mr}_{\text{S}}, \text{mr}_{\text{C}}, ...) \tag{3.113}$$

$$\text{mr}_{\text{ID}} := \bigcup (\varepsilon_M^{\text{ID}}, \text{rnk}, \varepsilon_M, \text{QUERYTERM}),\ (\varepsilon_M^{\text{ID}}, \varepsilon_M, \text{QUERYTERM}) \in \text{mrc}_{\text{ID}}, \tag{3.114}$$

$$\text{rnk} := \text{s}_{\text{ID}} \cdot \frac{|\text{mt} \cap \text{m}|}{|\text{m}|} + \text{s'}_{\text{ID}} \cdot \frac{|\text{mt} \cap \text{cm}|}{|\text{cm}|},\ \text{m} = \varepsilon_M.\text{files},\ \text{cm} = \varepsilon_M^{\text{ID}}.\text{files} \tag{3.115}$$

$$\text{s}_{\text{ID}}, \text{s'}_{\text{ID}} \text{ cf. Appendix A (p. 195)}, \text{mt cf. equation (3.116)}, \text{ID} \in \text{REC-IDS}$$

The operation we provide in equation (3.113) relies on some extra information while processing the model recommendation candidate sets given by subsection 3.5.5 and building model recommendations as given by equation (3.110). This extra information comprises a set of matching terms mt and a scaling denoted as scale_{ID} in equation (3.115). In addition, an operation related to the candidate ranking is included, because equation (3.115) is only based on the original model so far. Other than that, equation (3.114) relies on known information about candidate sets and expresses how the model recommendation sets are built. This eventually translates into in line 17 of pseudocode 3.6.

The first step in the ranking operation is to derive a set of matching terms wrapped in a model, which will support the matching of the given terms with model recommendation candidates or, to be more specific, their origin, as we explain later. Given a QUERYTERM

and context ($c \in \mathcal{C}$), we can semi-formally create a model of matching terms (mt). In doing so, we must bear in mind that m, TERMS, and $\mathfrak{Z}^{\mathtt{all}}$ with $\mathfrak{Z}^{\mathtt{del}}$ are part of the context.

$$\begin{aligned} \text{mt} :=& (\mathsf{E}_\mathsf{A}, \mathsf{E}_{cl}, \mathsf{E}_\mathsf{C}, \dots, \rho_{\text{eAttributes}}, \rho_{\text{eClassifiers}}, \dots, \mathfrak{R}), \\ & \mathsf{E}_\mathsf{C} := \text{m-words} \cup \text{t-words} \backslash \text{d-words} \cup \text{qt-words} \end{aligned} \tag{3.116}$$

$$\text{m-words} := \{\text{str} \in \text{STR} : \varepsilon_\mathsf{C}^{\text{str}} \in \text{m}.\mathsf{E}_\mathsf{C}\} \tag{3.117}$$

$$\text{t-words} := \{\text{str} \in \text{STR} : (\text{str}, \text{val}) \in \text{TERMS}\} \tag{3.118}$$

$$\text{qt-words} := \{\text{str} \in \text{STR} : \text{str} \in \text{QUERYTERM}.\text{normalizedSplit}()\} \tag{3.119}$$

$$\text{d-words} := \{\text{str} \in \text{STR} : \varphi_\gamma^{\text{str}} = \pi_d \leftarrow \bigcirc_{i=0}^{d \in \mathbb{N}} \pi_i \text{ of } \mathfrak{Z}^{\mathtt{del}}, \text{d} = 0, \dots, \mid \mathfrak{Z}^{\mathtt{del}} \mid\} \tag{3.120}$$

This model of match terms is semi-formally only, because we used some roughly provided functionality. We still opt for this, because it eases comprehensibility and our further explanations. Hence, we omit exact operations, which analyze the given QUERYTERM providing single strings in equation (3.119), or specific operations from a delete sequence ($\mathfrak{Z}^{\mathtt{del}}$) in equation (3.120).

With respect to comprehensibility, the first thing to note in equation (3.116) is the order in which the sets are constructed. This is of particular importance for the set-minus and the set added after that. In a sense, this overrides the removal and re-adds strings. Consider a model and its set of strings as given in equation (3.117). It does not contain any strings, i.e., names of elements, which were recently deleted, but the TERMS given by the context (c) might. Hence, these strings are removed by means of the set build in equation (3.120). In thinking of regular recommender systems, this expresses rejected items and should work in our approach unless a user opts otherwise. Therefore, the last set in equation (3.119) adds them again, though we do not describe how a QUERYTERM is subdivided into different terms, which are not keywords for the `Querys` (":wmodel"). In our overview, this is reflected in in lines 10 to 13 of pseudocode 3.6, although the set is neither built explicitly nor is the matching target clear at this point.

The matching target used in the second step of the ranking operation is the original model. The idea behind this is to take the value as a ranking basis for the actual candidate in the same way it was generated from a basis (cf. $\varrho_{gen}^{x-\text{SENS}}$ and $\varrho_{gen}^{\text{ID}}$). Thus, a value for the original model ($\varepsilon_\mathsf{M} \in \mathsf{E}_\mathsf{M}$) is first computed and then a scaling (s_{ID}) adjusts this value to the candidate type. In detail, the fraction of two set cardinalities (cf. equation (3.51) (p. 81)) is built and scaled. The basis for the nominator is the intersection between the matching terms and the original model, and the basis for the denominator is the original model, as expressed in equation (3.115) and later in line 14 of pseudocode 3.6. Hence, the value without scaling is between one and zero. Similarly, the actual candidate is treated in equation (3.115). Hence, the same matching terms are used for a fraction with the candidate, but then a different scaling ($\mathsf{s'}_{\text{ID}}$) is applied and the sum without scaling is between zero and two.

Finding the appropriate scaling factors for the original and candidate is an experimental task, as it is for regular knowledge-based recommender systems. For now, we set $\mathsf{s}_{\text{ID}} = 1$

and $s'_{ID} = 0.5$ without further explanation. In appendix A, we provide some pointers that might help develop scalars or functions that take into account the purpose of the recommender strategy under development. In general, we require the values to respect a certain order, and their summation is given as follows:

$$1 \geqslant \frac{1}{s_{EL}} \geqslant \frac{1}{s_S} \geqslant \frac{1}{s_C} \geqslant \frac{1}{s^+_{CH}} \geqslant \frac{1}{s^-_{CH}} \geqslant \frac{1}{s_G} \geqslant \frac{1}{s_{CAT}} \geqslant 0 \tag{3.121}$$

$$\sum \frac{1}{s_{ID}} = 1, ID \in \text{REC-IDS} \tag{3.122}$$

Altogether, we have introduced a ranking operation (ϱ_{rnk}) that derives a rank from contextual information and has some similarity to the term frequency (tf), but not to the inverse document frequency (idf). The former comes from how we work with the terms in the fractions of equation (3.113) and the latter is because we do not consider all documents, i.e., all `Models` of our knowledge library. A summarizing and more algorithmic illustration of our ranking is provided in pseudocode 3.6.

```
1  Set<ModelRecommendations> rank(Context ctx, QueryTerm qt,
2                                 Set<ModelRecommendationCandidates> mrc){
3    Set<ModelRecommendations> mr = new ... ;  // format: (rel, mrs, mrc, ...)
4    for (ModelRecommendationCandidates rc : mrc) {
5      for (ModelRecommendationCandidate r : rc)  {
6        // r is (εid_M, ε_M, QUERYTERM) with format (E_M, E_M, str)
7        // semantics: (candidateModel, originalModel, queryterm)
8        Model m = r.originalModel;
9        Set<String> mt;      // for matching terms with originalModel
10       mt.add(matchingTerms(ctx.model, m)) // currently edited model
11         .add(matchingTerms(ctx.terms, m))       // contextual words
12         .remove(mathchingTerms(ctx.Ξ^del, m)); // no deleted elements
13         .add(matchingTerms(qt.getTermList(), m)); // but qt's words
14       double rank = scale(r.candidate, m, mt.size() / r.size());
15       ModelRecommendation rankedRecommendation =
16             (r.candidateModel, rank, r.originalModel, r.queryterm);
17       mr.getRecommendationsSetFor(rc).add(rankedRecommendation);
18     }
19   }
20   return mr;
21 }
```

Pseudocode 3.6: Ranking Model Recommendation Candidates

As a final remark, the ranking we have introduced uses our understanding of granularity and relatedness in the same way that the original model provides the basis value and the generated candidate adds value on top. This might lead to the doubling of values in the case of complete model recommendation, but this is expected. Other, probably unexpected, behavior is: Consider element candidates with an original model, which is entirely part of the model kept in the context. This leads to a maximum score, as for the submodel and complete case. However, this is also the behavior we need for filtering.

Filtering: In terms of context-aware recommender systems, we started model recommendation production in subsection 3.5.4 with contextual-prefiltering and now conclude it with contextual-postfiltering [AT11]. Hence, we use our given context c ($\in \mathcal{C}$) and QUERYTERM to work on model recommendations mr $= (\text{mr}_{\text{EL}}, ...)$ ($\in \mathcal{MR}$). This means that we provide more details for the filtering operation (ϱ_{fil}) declared in equation (3.75):

$$\begin{aligned} \varrho_{fil} :=& \mathcal{C} \times \mathcal{MR} \rightarrow \bigtimes (\mathsf{E}_{\mathsf{M}} \times \mathbb{R}^{+}) : (\mathsf{c}, \mathsf{mr}) \mapsto \mathsf{r} \\ & \mathsf{r} := \{(\varepsilon_{\mathsf{M}}^{\mathsf{R}}, \mathsf{rnk}) : (\varepsilon_{\mathsf{M}}^{\mathsf{MR}}, \mathsf{rnk}, \varepsilon_{\mathsf{M}}, \textsc{queryterm}) \in \mathsf{mr}_{\mathsf{ID}}, \mathsf{ID} \in \textsc{rec-ids}\} \\ & \varepsilon_{\mathsf{M}}^{\mathsf{R}} := \text{ see equations (3.124) to (3.126)} \end{aligned} \tag{3.123}$$

The elements ($\varepsilon_{\mathsf{M}}^{\mathsf{R}}$) that make up the result set (r) in equation (3.123) have certain properties. These properties can be of external nature or stem from contextual information. The former is often a program setting and could be the maximum degree of allowed similarity or a minimum required model quality. The latter is a more precise use of the contextual information introduced in subsection 3.5.4, and could remove model recommendations that have been allowed so far, because they aid the derivation of further model recommendations or because they are generated automatically. Subsequently, we introduce the notation for result sets that satisfy the filter properties without providing operations.

Automatically "over"generated model recommendations occur, particularly for unmatched model quality (cf. subsection 3.4.4 (p. 101)) or for element recommendations (r_{EL}) in cases when this exact element was recently deleted. In both cases, we can conclude that this model recommendation is not wanted. This is why we use either model quality or words derived from our delete sequence ($\mathfrak{Z}^{\texttt{del}}$), as shown in equation (3.120) for filtering. The latter, denoted d-words in equation (3.124), limits the allowed elements as follows:

$$\begin{aligned} & \mathsf{r} := \{(\varepsilon_{\mathsf{M}}^{\mathsf{R}}, \mathsf{rnk}) : (\varepsilon_{\mathsf{M}}^{\mathsf{MR}}, \mathsf{rnk}, \varepsilon_{\mathsf{M}}, \textsc{queryterm}) \in \mathsf{mr}_{\mathsf{EL}}\} \\ & \forall \varepsilon_{\mathsf{M}}^{\mathsf{MR}} \in \mathsf{r}_{\mathsf{EL}} : (\gamma, \textsc{str}) := \varepsilon_{\mathsf{M}}^{\mathsf{MR}}.\mathsf{files}.\mathsf{E}_{ne}, \\ & \qquad\qquad \textsc{str} \notin \text{d-words (from } \mathfrak{Z}^{\texttt{del}}\text{) cf. equation (3.120)} \\ & \Rightarrow \varepsilon_{\mathsf{M}}^{\mathsf{R}} = \varepsilon_{\mathsf{M}}^{\mathsf{MR}} \end{aligned} \tag{3.124}$$

Note that sets adhering to specific quality levels look quite similar, but foster `MetaInformation` and the `semantics` contained in `VersionInfo` of a `Model`, as shown in figure 3.30 (cf. figure 3.27). We omit a more formal notation for this case, because the cascaded class structure, though well-structured for an implementation, makes the formula too incomprehensible. Further, it does not provide any additional benefit.

The equation introduced above is semi-formal in two respects. First, it uses a dot notation to refer to the model in the ranked model recommendation and to the named elements in the model. The same holds true for the deleted words, which we build in equation (3.120). Other than that, we use our common type symbol (γ) for an arbitrary type, because the type we operate on is of no interest here.

Turning to the maximal degree of allowed redundancy, consider the model currently being edited to already contain a model recommendation completely. This results in a

high ranking for this model recommendation without it providing any real benefit. We can treat this in one of two ways, both relying on a threshold for the *completion degree*, denoted by c-deg$_{\text{MAX}}$. The first alternative simply drops (or does not include) each model recommendation that does not adhere to $|\cdot| \leqslant$ c-deg$_{\text{MAX}}$, similar to equation (3.125). We use this as a fallback for the second case (cf. equation (3.126)), which leverages this model recommendation and replaces $\varepsilon_{\text{M}}^{\text{MR}}$ with its highest-ranked "neighbor" ($\varepsilon_{\text{M}}^{\text{X}}$). This is a model recommendation with the same original `Model` ε_{M} in r$_{\text{ID}}$, but a different first element in place of the result tuple, as shown in equation (3.126) ($\varepsilon_{\text{M}}^{\text{X}}$). This breaks the former ranking in terms of purpose by "boosting" the neighbor. The result set (r) is:

$$\text{r} := \{(\varepsilon_{\text{M}}^{\text{R}}, \text{rnk}) : (\varepsilon_{\text{M}}^{\text{MR}}, \text{rnk}, \varepsilon_{\text{M}}, \textsc{queryterm}) \in \text{mr}_{\text{ID}}, \text{ID} \in \textsc{rec-ids}\}$$

$$\forall \varepsilon_{\text{M}}^{\text{MR}} \in \text{r}_{\text{ID}} : \frac{|\text{m} \cap \text{mr.c}|}{|\text{mr.c}|} \geqslant \text{c-deg}_{\text{MAX}},\ \text{mr.c} := \varepsilon_{\text{M}}^{\text{MR}}.\text{files} \tag{3.125}$$

$$\Rightarrow \varepsilon_{\text{M}}^{\text{R}} = \varepsilon_{\text{M}}^{\text{X}},\ \begin{cases} \max\limits_{\text{rnk'}}(\varepsilon_{\text{M}}^{\text{X}}, \text{rnk}', \varepsilon_{\text{M}}, \textsc{str}) \in \text{r}_{\text{ID}},\ \varepsilon_{\text{M}}^{\text{X}} \neq \varepsilon_{\text{M}}^{\text{MR}} \text{ and drop used.} \\ \varnothing \text{ otherwise} \end{cases} \tag{3.126}$$

This filtering requires further explanation in two respects, one for typical use and one regarding the notation we have used. The most typical application of equation (3.126) is for "complete" model recommendations, because the original `Models` are the best matching results regarding contextual information and QUERYTERMS. Hence, they result in high scores and we could limit the given formula accordingly. However, we keep to the more general approach, though we should not expect many other cases in practice. This also has the side-effect of filtering element model recommendations if their contribution is already present in the currently edited model. With respect to the notation we have used, we omit a more formal introduction of how to drop a model recommendation that was used as part of a replacement. Instead, we rather sloppily write "and drop used" in equation (3.126). Similarly, we set the case otherwise, i.e., the case when no "neighbor" was found, to the empty set, though we should drop it entirely.

Although our approach is built to have independent UIs and recommender strategies, some filtering could be part of model recommendation production. First, the employed trigger method (τ) could be used, because we may expect proactive methods to benefit from smaller model recommendations when they are run after the creation of elements. For example, a created class results in recommending classes, or a created attribute results in recommending attributes, as in equation (3.109). However, in the case of many successive select operations, a proactive method benefits from larger model recommendations, e.g., achieved by ":iso" queries, which lead to complete model recommendations. Detecting such interaction patterns is beyond the current scope, but support for trigger-derived filtering can be developed from our concept of purposes for model recommendation candidate generation, as introduced in subsection 3.5.5.

Another goal in filtering is to introduce surprises, and an otherwise filtered model recommendation could be included at a fixed position in the result list of model recommendations. Alternatively, the select sequence ($\mathfrak{Z}^{\text{sel}}$) can bolster selection for this purpose. We mention these only as ideas, because we consider both as options for UIs,

which should offer and request them as properties.

Putting together subsections 3.5.4 to 3.5.6, we have all the necessary operations for model recommendation production (MP: ϱ_{ana}, ϱ_{gen}, ϱ_{rnk}, ϱ_{fil}), but, as we alluded to in equation (3.76), there is more to come. In particular, the candidate generation (ϱ_{gen}) offers options for building different recommender strategies, as depicted in figure 3.33.

3.5.7. Schema of Recommendation Production and Candidate Generation

Producing model recommendations follows the production sequences of operations we introduced by means of equation (3.76), and we illustrate recommender strategies in figure 3.33 that adhere to certain schema. On the one hand, these schema for model recommendation production can be *condition-adhering production* or *condition-altering generation*. Both foster the introduced operations to different extents, as we show below and summarize in figure 3.35. On the other hand, the schema can be on a higher level.

Condition-adhering schema for model recommendation production rely on given information and do not alter these conditions for reasons of potentially insufficient results (cf. table 3.13). The most straightforward schema is the "default production schema". This comprises a reactive trigger (τ), uses basic sensitivity (BASIC-SENS) for a given context (c), and performs ranking and filtering operations considering the given completion degree (c-deg$_{\text{MAX}}$). More precisely, we could call this schema *default reactive production* to contrast it with the other two condition-adhering schema, which are proactive. First, the *proactive-create production schema* builds on our default production schema, but comprises a proactive trigger (τ) for the case in which the most recent operation (in $\mathfrak{Z}^{\texttt{all}}$) was a create operation (in $\mathfrak{Z}^{\texttt{cre}}$). This information is used to produce model recommendations of element granularity that match the last-created element. For example, if the last operation was to create a class, then classes are recommended. Second, the *proactive-select production schema* also builds on our default production schema, but has a proactive trigger (τ) for the case in which the last operation (in $\mathfrak{Z}^{\texttt{all}}$) was a select (in $\mathfrak{Z}^{\texttt{sel}}$). Then, the recent sequence of select operations (in $\mathfrak{Z}^{\texttt{all}}$) is used for isomorphic sensitivity (EDIT-SENS), gaining complete model recommendations. The application scenario looks as follows: if successive selects or a drag-select are succeeded by a timeout, then complete model recommendation is recommended.

Table 3.13.: Condition Adhering Production Schema

Schema Name	Trigger (τ)	Leveraged	Generation (ϱ_{gen})	Filtering (ϱ_{fil})
Default Reactive	reactive	QUERYTERM	$\varrho^{\text{C}}_{gen} \circ \varrho^{\text{BASIC-SENS}}_{gen}$	c-deg$_{\text{MAX}}$
Proactive-Create	proactive	$\mathfrak{Z}^{\texttt{cre}}$	$\varrho^{\text{EL}}_{gen}(\circ\varrho^{\text{BASIC-SENS}}_{gen})$	c-deg$_{\text{MAX}}$
Proactive-Select	proactive	$\mathfrak{Z}^{\texttt{sel}}$	$\varrho^{\text{C}}_{gen} \circ \varrho^{\text{EDIT-SENS}}_{gen}$	

Condition-altering schema for model recommendation candidate generation is a means to generate more model recommendations and comes in two flavors, without considering

stemming or stop word removal. The first flavor comprises alternative generation operations, as introduced in this section, i.e., replacing one operation with a less restrictive one. The second flavor deals with queries to a given knowledge library in which the indexes are leveraged, i.e., replacing an index with fewer restrictions or a larger scope.

The *generation extension schema* approaches the problem of insufficient model recommendations by building on the sensitivity or granularity and relatedness (cf. table 3.14). The first, called *sensitivity extension generation schema*, extends the information used for generation. Hence, if a basic sensitivity (BASIC-SENS) does not generate enough candidates, the properties sensitivity (PROP-SENS) or terms sensitivity (TERMS-SENS) might. Note that this is possible because our generate operation (ϱ_{gen}) allows successive generations. In fact, this is the reason why the PROP-TERMS-SENS introduced in equation (3.99) is not really necessary, as we have explained already. The second, called *fallback extension generation schema*, uses more approximate than precise data for generation. Our example is the synonym sensitivity (SYN-SENS). The last generation extension schema, called *extent extension generation schema*, makes use of additional information provided by a knowledge library. Thus, if the candidate generation requires enhancement, the available sensitivity, e.g., grouped or categorized `Model`s, is considered. Note that this is not necessarily helpful for smaller granularity, and only makes sense for cross-links granularity or relatedness. Further, it is important to bear in mind that this might undermine the given purpose.

Table 3.14.: Condition Altering: Generation Extension Schema

Schema Name	Insufficient Result	Leveraged	Generation (ϱ_{gen})
Sensitivity Extension	$\varrho_{gen}^{\text{BASIC-SENS}}$	C	$\varrho_{gen}^{\text{TERM-SENS}} \circ \varrho_{gen}^{\text{PROP-SENS}} \circ \cdots$
Fall-back Extension	$\varrho_{gen}^{\text{BASIC-SENS}}$	QUERYTERM	$\varrho_{gen}^{\text{SYNO-SENS}} \circ \cdots$
Extent Extension	$\varrho_{gen}^{\text{BASIC-SENS}}$	KL	$\varrho_{gen}^{\text{CAT}} \circ \cdots \circ \varrho_{gen}^{\text{C}} \circ \cdots$

The last condition-altering schema concerns the indexes leveraged in querying a knowledge library (cf. table 3.15). Thus far, all sensitivity-related generation is built on a QUERYTERM such as "$\Psi(\mathrm{I}_{\text{WModel}})$, NAME" querying $\Phi_{\text{KL}}^{\text{QUERYTERM}}$ for a given NAME. This means that a term from the QUERYTERM must match a term, i.e., word, from a model. However, our knowledge library provides more indexes for querying, e.g., for the given `Model purpose` ($\Psi(\mathrm{I}_{\text{WPurp}})$) or its `name` ($\Psi(\mathrm{I}_{\text{WName}})$) and `description` ($\Psi(\mathrm{I}_{\text{WDescr}})$) (see equation (3.23) (p. 59)). Hence, the first schema, called QUERYTERM *index adjustment schema*, simply adds another QUERYTERM with the index replaced. In our example above, "$\Psi(\mathrm{I}_{\text{WModel}})$, NAME" is extended by "$\Psi(\mathrm{I}_{\text{WName}})$, NAME". Note that an index might not be suitable for every source of information, but here the "words in name" index is a possible addition. The second schema is a *context index adjustment schema*, which is well-suited to contextual information. Hence, a QUERYTERM derived, e.g., for PROP-SENS sensitivity, can be altered accordingly. This can hinder ranking, because the `Model`s might not contain rankable information. The reason is that the terms were found as matches for `Model purpose` or `description`, which are not yet considered in the ranking.

Table 3.15.: Condition Altering: Index Adjustment Schema

Schema Name	Suitable for	Adjustment
Queryterm Index Adjustment	QUERYTERM	$\Psi(l_{\text{WName}})$
Context Index Adjustment	C	$\Psi(l_{\text{WDescr}}), \Psi(l_{\text{WPurp}})$

The schema for model recommendation production introduced above are solutions for potentially insufficient numbers of model recommendations, but the opposite might also hold true. In other words, a result set could exceed a maximum number of possible model recommendations, and this is exactly the issue we approach in filtering.

In terms of regular recommender systems, contextual information, i.e., context-aware recommenders that leverage “neighborhood approaches”, is one way to extend a two-dimensional recommender to a three-dimensional system [AT11]. We illustrate this in figure 3.35 and note that granularity is not a “neighborhood”. This is one reason why we deliberately distinguish the extent of granularity from that of relatedness. Other than that, the dimension of the extent is discrete, whereas those of sensitivity and indexes are not. This is because of the extensibility of the latter. Other levels of extent are possible, but they require more effort by our knowledge library and other concepts. Compared to this, a new index for model features or sensitivity operation are quick changes.

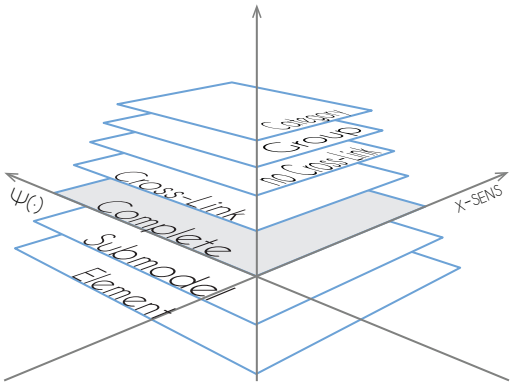


Figure 3.35.: Dimensions of Model Recommendation Production

Another important perspective and term for our approach stems from regular recommender system and is called the *dual mixed recommender approach*. For regular recommender systems, this is illustrated in figure 3.36 and shows two (for now) independent recommender systems. One is content-based, the other is an item-to-item recommender. Both operate following the steps introduced above, and finally combine their ranked lists. The notable point in this illustration is how the candidate set is fed to the item-to-item recommender. This is depicted with an emphasized line. The difference between our approach and the regular recommender system is denoted by the dashed lines. Whereas a regular recommender needs them, they are not necessary in our case.

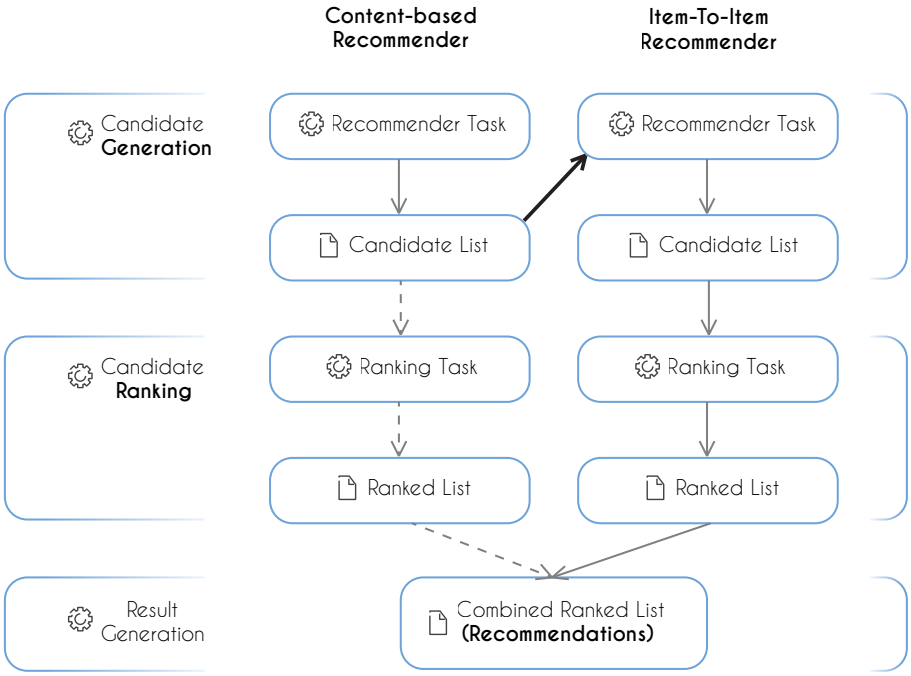


Figure 3.36.: Dual Mixed Recommender Approach Altered from [Nag13]

However, if we allow the composition of recommender strategies in figure 3.33, we can build a dual mixed recommender as depicted in figure 3.36 and require a job scheduling concept for recommender strategies. In subsection 3.5.3, we introduced the idea of scheduling concurrent recommender strategies and discussed the impact on our interaction model. Now, we realize this by requiring our recommender strategies to provide status information given by job scheduling. The top-level states for controlling are `enabled`, `disabled`, and `defect` as depicted in figure 3.37. Moreover, this figure shows the more fine-grained states that a recommender strategy can hold while data processing is `enabled`. In detail, a recommender strategy is `ready` if the initial checks on the setup were passed, e.g., database connections are validated. The recommender strategy remains in this state until it is requested to `search`. This puts it in a `running` state and, if everything works well, it signifies that the task is finished by moving to the `done` state. Otherwise, two alternatives are possible: first, the recommender strategy is prompted to `cancel` its current work by the controlling framework, or, second, the task `failed` for some reason, e.g., network timeout. In this case, a `reset` might help, returning the system to the `ready` state. However, if this is not possible, the recommender strategy can also reach the `defect` state. Note that this can also happen during initialization, e.g., if a required system is not reachable. As a final remark, note that composed concurrent recommender strategies are possible in our approach, because we provide a common foundation for ranking and can postpone filtering in the composing recommender strategy.

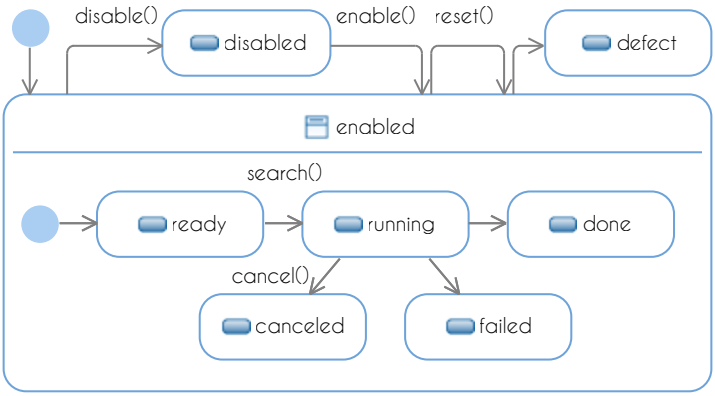


Figure 3.37.: Recommender Strategy States similar to [DGL14a]

3.5.8. Design Rationales and Observations

The approach for model recommendation production, which we introduced above, directly addresses the “Retrieval Challenge” described in section 1.2 (p. 6) and aims to provide model reutilization. At this point, we should state the grounds for achieving the major project goals, i.e., the E and S representing “easily and seamlessly” in HERMES. Note that these grounds are rather complex compared to those discussed in sections 3.1 to 3.4, because they build on these sections while combining and generalizing them, so we gain an approach in a cookbook or schema style with blanks that must be filled in (cf. appendix A (p. 195)).

The starting point for our rationales were typical sequences of recommendation production (cf. figure 3.36 left part) and asking what does not fit our case of model recommendation production. Bearing in mind that we are dealing with task-centric rather than user-centric environments and with item-to-item rather than collaborative filtering, we found that the available algorithms lack certain capabilities. This is because of the absence of semantic strength in the data used; we could leverage our knowledge library for this purpose and provide additional semantics and structures than in classic recommender systems.

However, we still need to provide the explicit reason why we should do so, and this lies in a slightly different goal for a model recommender system. Whereas classic recommender systems implementing collaborative filtering provide items based on similar user behavior, this does not make sense in our case, because in the extreme, we could end up with another identical model. Hence, we do not focus on the similarity of items at first, but on the item-to-item relationships we find in our knowledge library. In addition, we do not obtain much by analyzing user behavior, because long traces of modeling behavior are obviously less relevant than project information or actual contextual information, as we discussed earlier. Put together, we have the task-centric and item-to-item requirement for leveraging some (big) data.

Hence, the question is how much we can obtain from unstructured data: our answer, as the above discourse illustrates, lies in the semantically structured data leveraged from

our knowledge library. This enables our model recommender system to provide items even in the case of sparse results, which come from the fallback mechanism discussed for our model recommendation production schema.

But how did we find these schema? Certainly, the environment and the iterative approach we have developed played a role, but the central pieces of information were derived from implementing our experience and deploying several model recommender strategies. As a result, we found commonalities and common approaches for performing model recommendation production, which we also added to our dashboard (cf. section 4.5 (p. 156)). We also found that some variables for adjustment must remain as parameters, e.g., the completion degree introduced as c-deg$_{MAX}$. In addition, one of the main findings is that an ease-in is often necessary. In other words, an existing deployment environment, e.g., a directory of models for reuse, must be migrated to form a reasonable knowledge library. Note that we do not mean to say that a fully developed knowledge library is the only way! However, there is no harm in rolling it out incrementally, which requires support from model recommendation strategies, and our method of formulating schema (cf. appendix A (p. 195)) enables this incremental approach.

Unfortunately, this cookbook or schema requires a rather formal notation, but the upside is that it quickly translates in deployment scenarios. This holds true for both graph walk and model query mechanisms; the first is an approach for a given knowledge library and the second is an approach for granularity and relatedness.

Thus far, our explanations remain at the conceptual level for many aspects, with many real-world issues omitted. First, context analysis faces such issues. For example, graphical modeling often exposes element rearrangement. In this case, elements often appear in a select sequence ($\Im^{sel}$) without providing any benefit for model recommendation production. Certainly, such patterns should be detected and omitted. Second, it is commonly believed that a recommender like ours should support model fixing and refactorings [SU13; KMR13; RSA13]. Though we do not appreciate the term in modeling, we can formulate the necessary sequences and a recommender strategy leveraging our context. In particular, the delete and create sequences ($\Im^{del}$, $\Im^{cre}$) in combination with the currently edited model makes this possible. However, we wonder what the entropy of these changes might be. Certainly, the structure should be enhanced, but the content is not, i.e., no additional information (novelty) is gained.

Third, when a model recommendation is picked (cf. figure 3.34), the actual reutilizations takes place. This is not part of the model recommendation production, but the context component from figure 3.33 plays a major role in reutilization. This is because of the platform independence of models, which must be transformed into model operations applicable to the current editor. Further, some additional aspects need to be kept in mind. When an element recommendation is picked, the order of an application sequence is crucial. As created elements should be inserted properly, e.g., “connected”, the first operation should be a find operation that delivers the proper container. Only after this should the element be added and finally related. For example, reutilizing an E_A should first determine the proper E_C in the currently edited model, and only add it after that. Another approach is to build the element model recommendations employing the

submodel operation introduced in subsection 3.3.2 (p. 75). We can then create two models and a cross-link that can be applied as follows: (i) the element is created, (ii) the cross-link is applied. The above explanations did not consider this for simplicity and because it is an implementation detail. To the best of our knowledge, the latter approach is the most straightforward.

In addition, the ranking can be only a proposal, and there are approaches that might be worth exploring. First, degree-of-interest models are popular in other implementations [MMA14], e.g., Mylyn [BR11; KM05; KM06]. Second, analyzing content, as already indicated at the start of subsection 3.5.6 for tf-idf, may be worth exploring [Arb11].

3.5.9. Related Work

The roots of recommender systems can be traced back to Information Management Systems and Decision Support Systems (DSS) in the 1980s [Spr80], and some ideas might be worth revising, although, for the sake of brevity, we keep to recommender systems and the recent terminology. Moreover, we look into more recent frameworks without contrasting them with DSS. Until recently, most recommender systems were tied to the web, e.g., as part of commercial systems [SKR99], such as Amazon's recommenders [LSY03]. However, recommender systems specific to software engineering have emerged to assist software developers in a wide range of activities, including code reuse [RWZ10]. Most of these systems are integrated in an IDE and suggest software artifacts, such as code snippets, while focusing on "you might like what similar developers like" scenarios. A detailed discourse is provided by Happel and Maalej [HM08a], and we subdivide our discussion into source code related recommender systems, modeling related recommender systems, and other recommender systems.

Source Code Related Recommender Systems: Five code recommenders and recommender systems have inspired us. First, the project *Code Recommenders* by Bruch is a recommender system for the Java programming language and is integrated into the Eclipse IDE [Bru12; BSM08; BM08b; Bru08; WKB09; Ecl14a]. It comprises various intelligent code completion engines and documentation providers. For example, its intelligent call completion recommends only methods that are most likely to be called at the current editing position (cf. figure 1.1 (p. 3)). Code Recommenders' dynamic template completion takes this to the next level by recommending a complete sequence of method calls. To this end, it uses available open-source code repositories to analyze common code structures. These code templates can serve as additional documentation that quickly shows how an API can be used, thus saving developers' time when using APIs they are not familiar with. Second, *SnipMatch* recommends common code snippets, similar to Code Recommenders' dynamic templates. However, the developer queries the system describing the task they want to accomplish (cf. figure 3.38) [Ecl15b]. SnipMatch is now part of the Code Recommenders project. Third, *Code Conjurer* by Hummel, Janjic, and Atkinson is a recommender system that uses code-search engines to deliver high-relevance software-reuse recommendations with minimal disturbance to a developer's workflow [HJA08]. It seamlessly integrates code search and reuse functionality

into the Eclipse Java development environment, thereby allowing developers to search for reusable code, e.g., by defining unit tests (test-driven search). Code Conjurer delivers code recommendations as results that satisfy certain tests and even generates adapter classes to match the interface specified in the tests, if needed. Altogether, it uses a source code search engine based on Merobase by Janjic et al. [Jan+13]. Fourth, Strathcona is a proactive recommender system developed by Holmes, Walker, and Murphy for source code examples [HWM06]. Their approach takes the current editing context into account to perform a structurally approximate context matching with the aim of finding full-text examples that support developers. Finally, WitchDoctor by Foster, Griswold, and Lerner can detect editing patterns and recommend refactorings [FGL12].

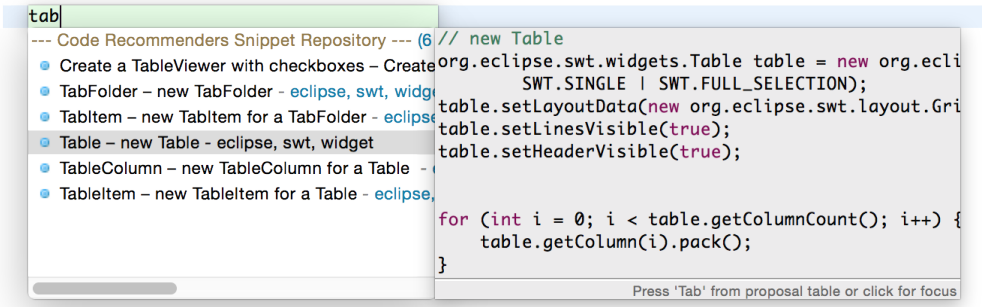


Figure 3.38.: Querying SnipMatch [Ecl15b]

Modeling Related Recommender Systems: Modeling support exceeding simple reuse [SA13], with recommender systems and the like, can be subdivided into three groups of approaches. The first is based on logical programming, namely Prolog. White and Schmidt presented a framework for domain-specific modeling languages on a conceptual level [WS06]. They focus on establishing domain-specific knowledge bases and algorithms so they can work in what they call “combinatorically challenging domains”. For example, they semi-automatically map logical models to deployment models for car electronic control units using a Prolog knowledge base. As the modeling language in their example, they use AUTOSAR. In contrast to their solution, we do not seek to map elements of different models for modeling support, but provide additional content. Another solution that uses Prolog is that of Sen, Baudry, and Vangheluwe [SBP07; SBV08; SBV10]. They demonstrate their “partial model completion” with finite state machines and offer a brief methodological overview of how to synthesize a model editor from a metamodel in declarative form, constraints, and a visual syntax. Their knowledge base holds a constraint logic program derived from several models and is meant to deduce recommendations. Our work differs in that we do not have a knowledge base we can draw conclusions from. Further, we do not aim to synthesize model editors, but extend existing editors with our approach and use expert-validated models.

The second group of approaches uses the Triple Graph Grammars (TGG) proposed

by Schürr, although these were initially meant for model-to-model transformations or model synchronization [Sch95; KW07]. For example, Mazanek, Maier, and Minas use TGGs to provide syntactic auto-complete functionality for Nassi–Shneiderman diagrams [MMM08b]. They use an approach that can transform "possibly incomplete graphs" into graphs that are members of a graph grammar [MMM08a]. This is useful for completion mechanisms in diagram editors in terms of the auto-complete functionality. Their example demonstrates this using two isolated statements that need to be composed to form a correct diagram. Three possible solutions are given, and these could be called recommendations. They benefit from TGG characteristics, i.e., all transformations are bidirectional and allow batch and incremental transformation. For us, the question is what might be "possibly incomplete"? Looking at syntactic links in our enhanced knowledge graph and the adjacent node, the stored syntactic link plays exactly this role. Furthermore, the adjacent node and its related model are the second part. Hence, we do not need TGGs, but can start including the syntactic link to gain an extended model.

The last group of modeling support with recommender systems and the like concentrates on modeling languages rather than the underlying concepts. First, a UML-related recommender system proposed by Kuhn produces "meaningful names" [Kuh10]. These are meant for naming methods and other textual elements in UML models, and rely on numerous other language processing systems, "real" recommender systems, or supporting systems that produce names for (mostly) source code editors, as mentioned above. Another UML-related modeling support system built on patterns is that developed by Kuschke, Mäder, and Rempel [KMR13] and Kuschke and Mäder [KM14]. Their rather general 38 modeling activities are made up of 19 modeling patterns that, when started, can be detected by their system and automatically completed. An example for a modeling activity is "Replacing an association between two classes by an interface realization (ap6)", which needs to be tailored before it can be applied to a modeling canvas. Hence, they offer editing for the remainder of sequences before it can be applied. Altogether, their approach is extremely similar to that offered by our strategies. The significant difference is that we offer more specific knowledge, such as design patterns and semantically meaningful models. Hence, we can build real recommender systems because our recommendations add genuinely new content and do not just restructure the model at hand. Still, we could run their system in our framework, but their realization is not freely available. Next, Palma et al. presented a recommendation system for supporting their design pattern recommender [Pal+12]. Their approach facilitates Basili's goal question metric [Bas92], and they define which GoF pattern (goal) [Gam+95] should be recommended according to the refactoring statements (question), and found their reasoning on metrics. Another UML-tailored approach that helps modelers is "model assist", as described by Steimann and Ulke [SU13]. This is very similar to the code- and content-assist methods for source code, and is meant to lead to syntactically correct models or fix malformed models. For example, circular inheritance is considered to be illegal. To detect such cases, model completion is formally defined to the extent that a constraint solver can detect potentially malformed models, meaning that editing can only lead to well-formed models. Hence, the approach itself does not consider semantic

contributions to a model. In other words, a new class representing a new concept could never be suggested by the approach, but this is exactly what we are aiming for and providing with our model repositories. Berardi, Calvanese, and Giacomo also worked on the correctness and soundness of UML class diagrams, for which they used description logics [BCG05]. Again, modeling support is provided, though this is computationally expensive . Similarly, the approach described by Queralt and Teniente works on aspects like redundancy or class liveness to leverage OCL constraints [QT06].

Second, more extensive research regarding recommender system and the like in terms of modeling languages was conducted for business process models and BPMN. However, Michael Fellmann et al. state that “recommendation systems [...] have not been exploited for business process modeling although implementation strategies have already been suggested” [Mic+15]. This means that no real-world deployment has been undertaken as of 2015. Still, they propose a requirements catalog for business model recommender systems [Mic+15]. These requirements are more or less congruent with our findings, but we aim for more flexibility in terms of architecture [GL13; DGL14a]. While we both aim to “Ensure recommendations with a high semantic quality” and provide “Multiple recommendation strategies”, or “Multiple ways of displaying recommendations”, flexibility in regard of contextual information is not their focus. The closest to that would be the “Compatibility to existing tools and languages”. The abovementioned Mazanek, Maier, and Minas extended their approach from Nassi–Shneiderman diagrams to work with TGGs and business process models in BPMN 2.0 [MM09]. Here, they calculate potential combinations or completions for model fragments. They inspect these as graphs and provide so-called hypergraph patches, which comprise merge and add actions. For example, a gateway with one branch modeled is completed by adding another branch and merging the gateways, i.e., adding a BPMN activity and connecting it to the opening and closing gateways. Wieloch, Filipowska, and Kaczmarek presented an approach and a prototype for semantic autocompletion [WFK11]. They analyze the editing context with a tool they call FragmentMatcher, and use this information to derive potentially semantical autocompletions. Certainly, previously fragmented and indexed models are required to allow FragmentMatcher to find a MatchSequence. Altogether, their architecture is similar to our recommender search strategies and model mining. However, we opted for the separation of concerns and built distinct frameworks for model mining and model reuse. Additionally, we aim for real recommender systems in our recommender search strategies. Although they never mention this term, their approach works almost like a recommender system. Hornung, Koschmider, and Lausen use signatures called virtual documents, which comprise tags or semantic annotations, to find models in repositories [HKL08]. Therefore, they first extract keywords from existing models to enable querying through their query interface. Moreover, they facilitate scoring to identify the best matching model, which makes their approach very similar to recommender systems. The only part they do not name is context consideration, but they describe how “the ranking mechanism changes [...] once [...] modeled process elements [are] in [the] workspace”. They do not elaborate on the architecture of their solution. Koschmider, Hornung, and Oberweis deal with the novice issue in modeling by providing a recommender-based

editor that offers strong interaction alternatives and metrics [Kos07; KO10; KHO11]. An autocompletion mechanism for executable business process models was proposed by Born et al. [Bor+09]. This undertakes context analysis by means of a Business Functional Ontology and checks for pre- and post-conditions in logic formulas for the potential completions. Furthermore, it performs a nonfunctional evaluation of availability and quality for the proposed services.

Third, Simulink has been a playground for recommender systems in model-driven development, and research has been conducted in this field by Heinemann [Hei12b; Hei12a]. He uses public Simulink model libraries and feeds their models to his recommender system. Comparing the results of collaborative filtering and association rules indicates that the former works better in his case.

Other Aspects of Recommender Systems: Some parts of our contribution are architectural, because not much attention has yet been paid to these aspects. For example, Fabiana Lorenzi, Ana L. C. Bazzan, and Mara Abel and Wohltorf, Cissee, and Rieger discuss architectural aspects for recommender systems, but only for data sources, i.e., knowledge libraries [FAM06; WCR05; Woh+04]. Both use agents, i.e., multi-agent systems, to gather data, and this is somewhat similar to our recommender search strategies. However, in our case, they are passive and meant to produce recommendations. This means they should perform steps including ranking and filtering, as recommender systems should do. Moreover, our solution is flexible in regard to UIs and contexts.

A more top-level editing approach is provided by the recommender system presented by Murphy-Hill, Jiresal, and Murphy [MJM12]. They monitor interactions with the Eclipse IDE, e.g., “Open Editor”, and then analyze them and recommend possible next steps if they are available as Eclipse commands. Our approach differs in respect of the types of commands being used. While Murphy-Hill, Jiresal, and Murphy work with declarative commands registered to the Eclipse IDE/Framework, we only use EMF commands, which are designed for model editing purposes. Furthermore, their mining and ranking algorithm needs user editing histories, which are not necessarily transferable between users. We aim to provide models, i.e., sequences of EMF commands, which we can use to “automatically” edit the current model.

3.5.10. Summary of Reusing Models

The previous section leveraged the concepts introduced before in several ways to build recommendation concepts for model reuse. Therefore, recommender systems for software engineering (RSSE) and model recommendations were discussed, and we explored how our knowledge library can foster model recommendation production. We subdivided this into recommendation operations (equation (3.71)), which can form a model recommendation production sequence (MP cf. equation (3.76) and figure 3.39). In more detail, production was subdivided into operations to analyze the provided data (ϱ_{ana}), generate model recommendation candidates (ϱ_{gen}), rank these candidates to become model recommendations (ϱ_{rnk}), and eventually filter these model recommendations (ϱ_{fil}) to produce a final set of results. In these respects, we introduced sets representing

the information at hand. For example, the results of model recommendation generation were denoted by mrc ($\in \mathcal{MRC}$ equation (3.88)) and the results from ranking were written as mr ($\in \mathcal{MR}$ equation (3.110)), so the result set could be r. These operations can be concatenated to read from right to left, which is contrary to the illustration in figure 3.39.

Many of the observations in developing this approach are based on the constraint dimensions of model recommendation production. Namely, these are the UI, available data, and permitted scope (table 3.10). They affect model recommendation properties called the extent, granularity, relatedness, sensitivity, and impact. In particular, the extent is subdivided into granularity and relatedness, which are important for model recommendations (table 3.11). This is because of the interaction model we introduced to provide ideas for triggering methods, called reactive and proactive (cf. figure 3.39). This information, denoted τ, was also contained in our understanding of context ($c \in \mathcal{C}$ equation (3.78)), which also comprised the currently edited model, the editing sequence (e.g., $\mathfrak{Z}^{\mathtt{all}}$, $\mathfrak{Z}^{\mathtt{cre}}$), and other contextual information. Other observations were at a higher level, and we introduced schema for model recommendation production and generation. These are either condition-adhering or condition-altering (tables 3.13 to 3.15). The latter fall into schema for model recommendation generation extension and schema for model recommendation index adjustment. Finally, we discussed the reutilization of model recommendations.

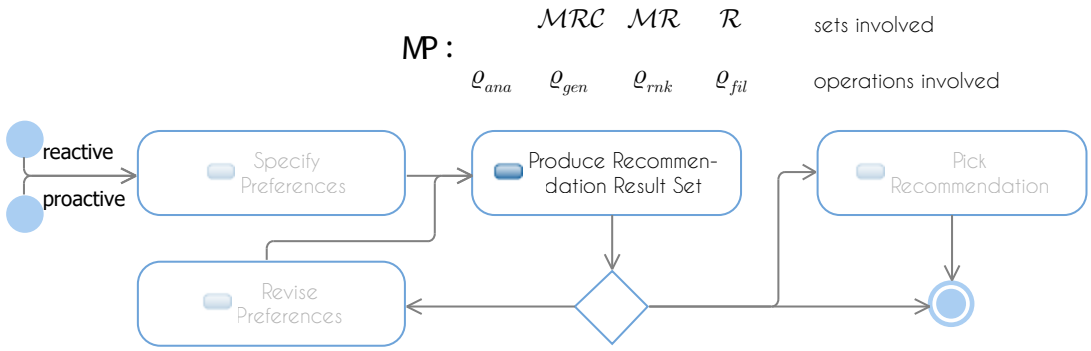


Figure 3.39.: Recommendation Production Overview (cf. figure 3.34 and equation (3.71))

3.6. Summary

Admittedly, this chapter deserves an extensive summary of the numerous operations, concepts, and ideas developed. We opt otherwise and postpone a full discussion until chapter 4 (p. 151), because we require a tailored version of a summary to begin with the concepts required for a realization. Still, we provide an overview of this chapter in figure 3.40.

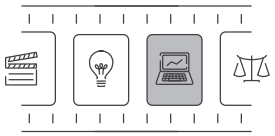
Figure 3.40.: Summary of Harvesting, Storing, Evolving, and Reusing Models see figure 3.21 (p. 93), figure 3.12 (p. 72), figure 3.31 (p. 116), and figure 3.39 (p. 149)

4 HERMES

If you think good architecture is expensive, try bad architecture.

Brian Foote and Joseph Yoder

Contents



The project name “Harvest, Evolve, and Reuse Models Easily and Seamlessly”, abbreviated as HERMES, inspired the name for our software prototype. Similarly, the ideas and concepts developed in chapter 3 (p. 31) were transformed into this software prototype, matching the respective parts of its conceptual architecture. For easier comprehension and because of the lack of an overall summary for the previous chapter, we reiterate the concepts above while concentrating on the aspects vital for a software prototype. We then go into more detail concerning the realization.

The root for our concept is operation-based models ($\mathfrak{Z} \in \mathcal{M}^{\mathfrak{Z}}$) and we provide a notation for constructing models (m) as sequences of operations (σ) made up of operations, e.g., π_c for create or $\varphi_\gamma^{\text{QUERYTERM}}$ for find. Further, we have introduced a knowledge library (KL) for storing models, wrapped in `Models`. These are enhanced with `MetaInformation`, interlinked by `Connectors`, and organized by `Categorys` as well as `Groups`. Consequently, a knowledge library has a graph structure. We can operate on it by indexing (I) and querying (Φ) elements with simple and complex indexes and QUERYTERMS. We obtain models wrapped in `Models` for our knowledge library by harvesting the models that are currently being edited. This happens in several steps: finding sets of already known elements ($\mathcal{K}$) from a knowledge library, building submodels (s) and cross-links (cl) between them ($\wr\!\wr \mathsf{s}_i$), and storing them. As we do not consider harvested models to be perfect for reuse, we allow them to evolve in a knowledge library. Therefore, we assign one of three quality stages to `Models` denoting sketchy, provisional, and high quality as `vague`, `decent`, and `fine` reusability. All this builds on a quality model with related metrics combined with an evolution stage automaton, which is partially automatable. The parts that are not automatable are supported with a special lightweight review mechanism. The final reuse requires models to be found and provided by our knowledge library, which is supported by approaches adapted from knowledge-based content recommender systems. Such model recommendation production follows processing steps called analysis, generation, ranking, and filtering. We have provided an operation each, denoted ϱ_{ana},

ϱ_{gen}, ϱ_{rnk}, and ϱ_{fil}. As intermediates, we introduced model recommendation candidates mrc ($\in \mathcal{MRC}$) as outcomes for ϱ_{gen}, model recommendations mr ($\in \mathcal{MR}$) as outcomes for ϱ_{rnk}, and results r as the final outcome. Finally, we have discussed schema for model recommendation production and generation as well as the impacts of the trigger (τ) and editing sequences (e.g., $\mathfrak{Z}^{\mathsf{cre}}$).

4.1. Conceptual Architecture

The conceptual chapter repeatedly referred to the underlying project overview from figure 1.8 (p. 12) for structuring its content in sections. We can now use this overview for bridging concepts to our software architecture. Figure 4.1 shows a version of our project overview with some parts faded and additional text. The text next to each respective conceptual part allows navigation within our realization, because we place features, fragments, and plug-ins in respective packages. Hence, the discussion of our realization follows this schematic. For example, `.store.mdf` comprises all realized "storing concepts" discussed in section 3.2 (p. 48) and alluded to in introducing figure 3.5 (p. 49). The exceptions to this are "operation-based models" from section 3.1 (p. 32), which are wrapped and adjusted "EMF Edit Commands", e.g., in section 4.5.

In bridging the concepts to realizations, one thing is striking: the change of terms we have undertaken. Whereas the conceptual level refers to "storing" models, the realization of the framework requires `.mdf`, which stands for "model data framework". Hence, the packaging denotes content `store.mdf`. In general, this change of terms is attributed to applied approaches. This becomes more apparent when looking at our "harvesting" of models, which is implemented as a framework denoted by `.mmf`, short for "model mining framework". Note that "evolving" the models remains the same in `.mef`, but "reusing" the models is realized as a "model recommender framework", abbreviated as `.mrf` in package `reuse.mrf`.

Figure 4.1.: HERMES Overview and Top Level Packages cf. figure 1.8 (p. 12)

With the given conceptual architecture, we can turn to the use cases, which we introduced as requirements for our project in figure 2.1 (p. 25). Other than the use cases for each framework, certain constraints hold true for each framework. General requirements are manifested in the project name "HERMES" and require the prototype to be (1) easy to use and (2) seamlessly integrated (cf. figure 1.4b (p. 6)). In other words,

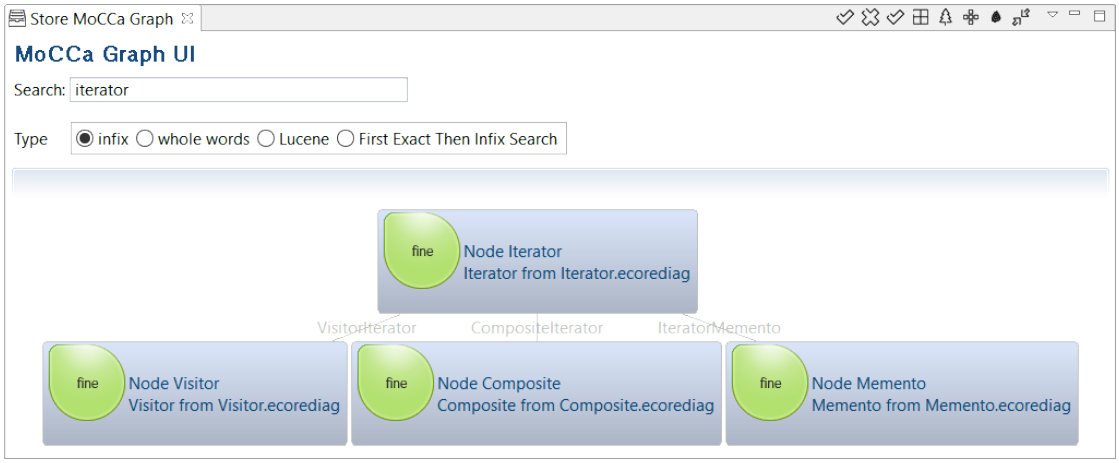


Figure 4.2.: Model Store Browsing the Knowledge Graph

we strive for simplicity, comprehensibility, and usability. Further, we require our prototype (3) to offer extendability concerning the architectures of respective project parts, i.e., for the data framework shown in figure 3.5 (p. 49), for the model mining framework in figure 3.14 (p. 74), for the model evolution framework shown in figure 3.23 (p. 95), and for the model recommender framework depicted in figure 3.33 (p. 118). Altogether, the prototype we will introduce incorporates some of the experiences described in the context of this document. For example, we have already summarized some experiences for our model data framework [GL13; TGL13], model evolution framework [Rot+13; Gan+13; Gan+16], and model recommender framework [DGL13; DGL14a; DGL14b].

4.2. `.store.mdf`

The backbone of every recommender system is the data it can use for model recommendation production, and one part of this is data organization as a knowledge library, as introduced in section 3.2 (p. 48). Hence, the major requirement is to provide potentially beneficial data for model recommendation production, e.g., through an API (Φ, Ψ), and simple editing functionality interlinked with our model evolution framework.

At the time of writing, there are two realizations in service as part of `.store.mdf`. The first is called MoCCa, which is an acronym for "*Mo*del *C*ombination and *C*omposition V*a*ult". Figure 4.2 shows a GUI for searching and exploring the knowledge graph. In addition, it enables (`Category`, `Group`, or `Model`) browsing of editing properties or contained models, and can relate different properties and terms. However, the more important functionality is API accessibility, because this is what the recommender strategies are built upon. This is realized in a service provider manner, so one entry point unveils functionality for storing, editing, and querying. In brief, the realization employs a Neo4J graph database with a Lucene search engine [RWE13; Mul12] and git version control system [CS09; LM12]. The second realization available as part of `.store.mdf` is called

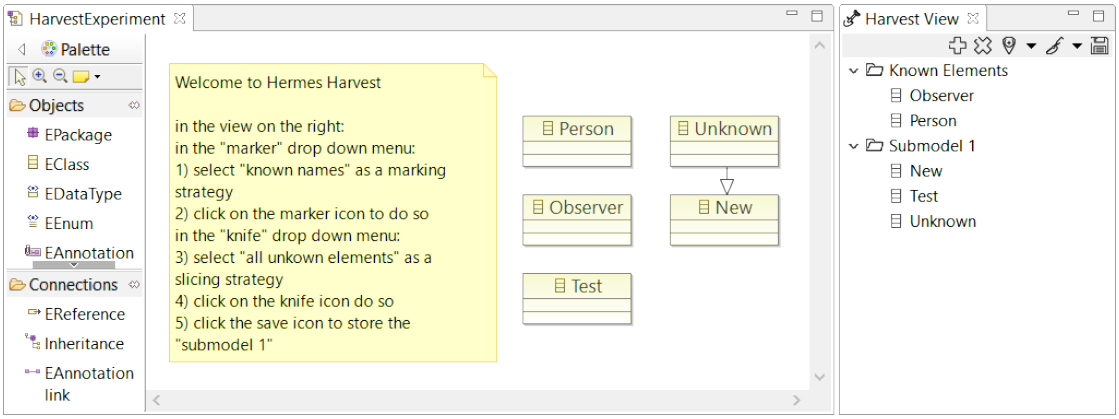


Figure 4.3.: Model Mining for Models with Known Elements

store and offers roughly the same functionality as a distributed system derived from figure 3.6 (p. 51). Therefore, we only mention a few realization aspects. First, the resource strategies for the generated parts of the knowledge library are not tightly linked to graph databases anymore and allow relational databases [TGL13]. Similarly, the version control system is not bound to git and information retrieval is not tied to elasticsearch [GT15].

4.3. `.harvest.mmf`

The other backbone of a model recommender system is the actual data available for model recommendation production. We have introduced an approach that supports users while harvesting models (see section 3.3 (p. 73)). The main requirement was to provide suggestions of what to store for reuse, but identifying reusable parts in a model under consideration is a rather subjective matter. Thus, the focus of our approach is threefold. First, already known elements should be identified; second, a model under consideration should be subdivided into beneficial parts; and third, these parts should be integrated in a knowledge library. However, apart from all this tool support, a modeler must always have the last word and provide the required `MetaInformation`.

Having a closer look at the prototype, the abovementioned parts belong to three different packages [Sew13]. First, identifying known parts belongs to our package for markers in `harvest.mmf.marker` and can be of different granularity. Second, splitting the remaining parts is done by a selected splitter, which originates in `harvest.mmf.splitter`. Generally, the splitting realizes a graph clustering algorithm, as introduced in subsection 3.3.2 (p. 75). Third, storing the identified clusters is part of the saver in `harvest.mmf.saver`, and this is tightly linked to possible persistence, as introduced in section 4.2. Regarding UIs, figure 4.3 shows the harvesting view on the right, with a tool bar offering add, remove, mark, split, and save operations (from left to right). Certainly, elements can be rearranged, and with meta-data provided and everything persisted in a knowledge library, the model evolution framework takes over for further processing.

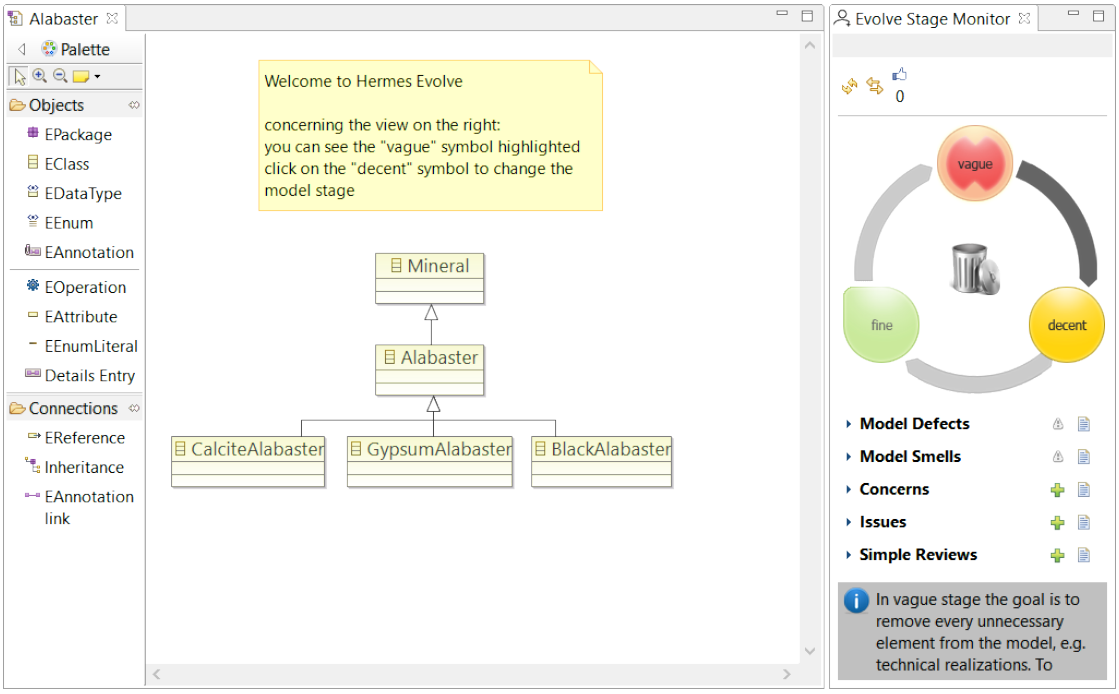


Figure 4.4.: Model Evolution with Alabaster Model and Stage Monitor

4.4. .evolve.mef

For our approach, the model evolution framework is considered an add-on that has special requirements, because operation-based model recommendations would not fall apart if evolution support was missing. However, this places extra requirements on its realization: any extra effort for users getting familiar with the prototype must be avoided. This means simplicity and supporting users in every possible way are the highest priorities. Hence, we opted for a prototype with a guidance mechanism to help modelers through the stages of model evolution, and use a restricted representation of our evolution stage automaton from figure 3.27 (p. 103) in our prototype (cf. figure 4.4). This was done in recognition of the use cases introduced in figure 2.1 (p. 25).

We call this simplified quality stage model the stage monitor, and use it to encompass additional information. We have already mentioned that stages also come with coloring, but now the deprecated flag finds its way into the stage monitor. If activated, the model is frozen in its current stage, which is still shown, and the deprecated symbol is highlighted. If deprecation is not activated, the current stage is highlighted along with the possible transitions and whether a stage is pending, i.e., if a stage requires manual confirmation. Further, information about measures subdivided into defects and smells are present. Recall that the former are errors in models similar to code smells, which may require improvement [Fow99]. Finally, we present information about reviews and their respective concerns and issues. It is simple to create new reviews limited to a desired type, as

shown in figure 4.5, and assign certain existing concerns or issues.

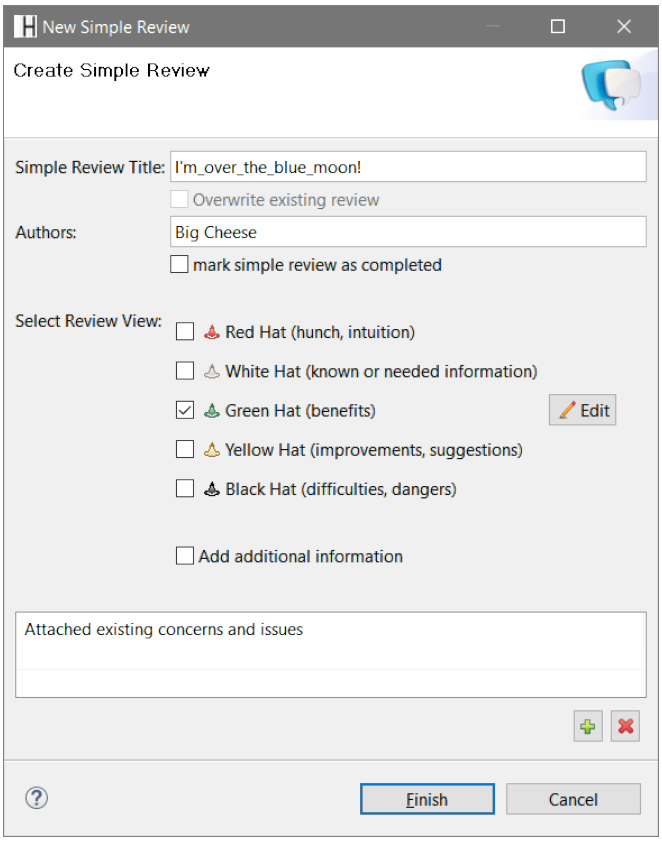


Figure 4.5.: Simple Review

All this serves to increase the guidance provided to modelers, as do the numerous tool tips, hints, information boxes, and wizards. They are all kept short, informative, and constructive, so violations of metrics can be resolved in a guided fashion. Further, modelers receive continuous feedback on violations (proactively in terms of our recommender systems) without it being obtrusive. Called proactive quality guidance [Gan+13], this should prevent violations and defects from piling up, which could be perceived as annoying and result in issues being ignored.

Concerning our implementation, the concept is depicted in figure 4.11, which shows how the `StageMonitor` holds the `MetricManager` and the `ReviewManager` together. The former calculates metrics for the model under consideration. Note that this allows for extension by new `ConcreteMetrics`, all of which are located in a top-level package called `evolve.mef.metrics`. The latter is responsible for `Reviews` and is potentially extensible, but introducing another `ConcreteReview` next to the existing ones from figure 3.28 (p. 105) should be well thought through. Other than that, the details provided in section 3.4 (p. 94) already imply many implementation details, e.g., how to realize a stage monitor with quality gates. Finally, note that, compared to figure 3.23 (p. 95), we do not describe the use of versioning, i.e., a version control system, because it is shared with the model data framework. However, we do allow resource strategies in a top-level package called `evolve.mef.vcs`.

4.5. `.reuse.mrf`

The essence of our prototype is contained in our model recommender framework, which was targeted at some functional and non-functional requirements. We introduced the

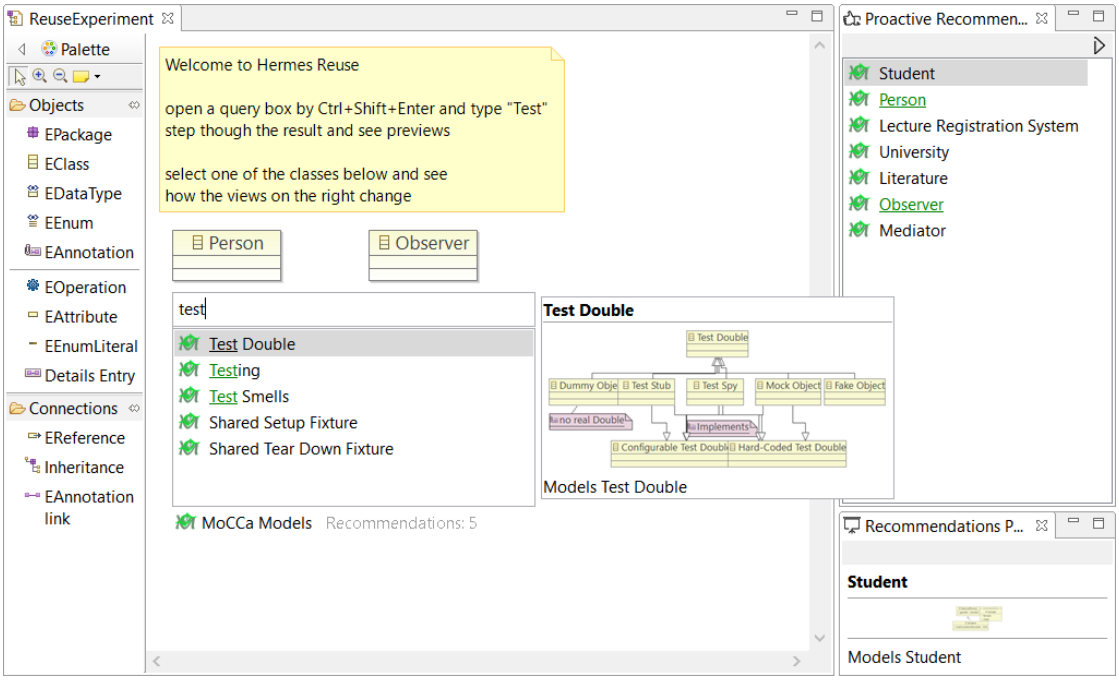


Figure 4.6.: Model Reuse with Proactive View and Reactive Query Box

former as use cases in figure 2.1 and implied the latter as constraints in section 3.5 (see also [DGL14a]). Hence, we pick up and summarize the non-functional requirements: Figure 3.33 (p. 118) shows potentially many recommender strategies, and we have already mentioned that each can use different sources of data. All that matters is that the framework needs no information about the object content except for how to apply it, so a context component can be integrated. Therefore, the recommender strategies that use these data backends should have the freedom to realize different algorithms according to the purpose, as introduced in subsection 3.5.5. This implies that different context components must be leveraged for querying and inserting in model recommendation production [DGL13], as well as different UIs [DGL14b]. Hence, we strive for a flexible architecture, because we found that the “[a]rchitectures of the surveyed [recommender] systems are inflexible and do not allow for extensions” [HM08a]. Finally, usability is key again, and we aim for non-blocking UIs that require threaded recommender strategy runs started by a job framework (cf. figure 3.37 (p. 142)).

Examples for the extension points are as follows: First, user interfaces are often graphical, so a `UIStrategy` can be a query box, as shown in figure 4.6 for reactive triggering (τ), or it can be a view that proactively updates following timeouts, as shown in figure 4.6 on the right. Second, contexts can monitor or access editors to provide a generalized format of information per editor, as defined in subsection 3.5.4 (p. 123). Thus, no matter whether an EMF, EcoreTools, Graphiti, or Sirius Editor is used, a context component performs the proper analysis because it is built for this dialect of modeling

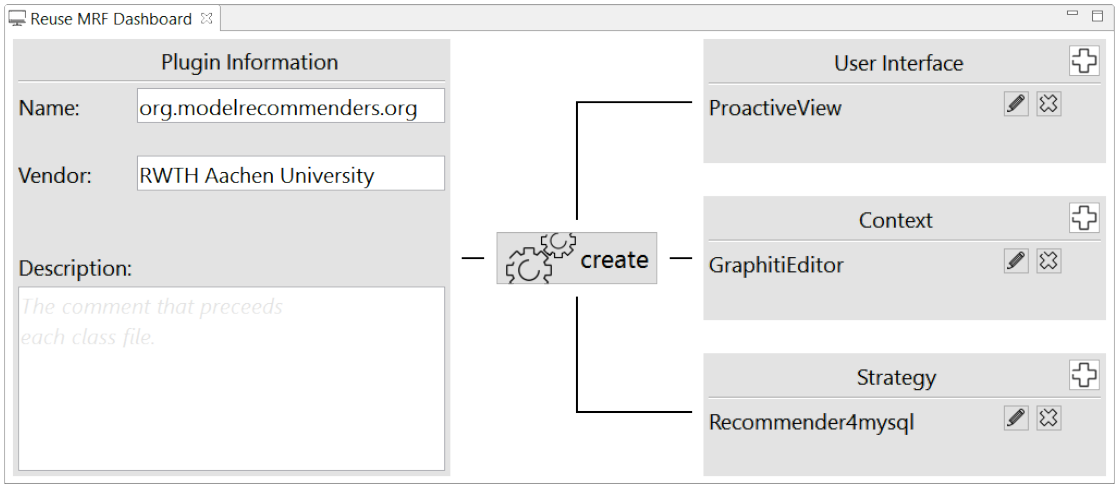


Figure 4.7.: Model Recommender Framework Dashboard similar to [Sch13]

language or representation. Further, it knows how to handle recommendations, if picked. That is, how to apply a model in operation-based format. Third, recommender strategies can be algorithms that fulfill a purpose while leveraging a knowledge library, as in our case, model repositories such as ReMoDD [FBC06] or MOOGLE [LFW12], or other data backends.

An implementation realizing the abovementioned requirements is sketched in figure 4.12 [DGL14a; Dyc12b]. The essence of it lies in the classes of `Recommendation` and `RecommenderSearchStrategy`. They build the framework hotspot (cf. [Pre96]), allowing the framework to be extended. The responsible extensions, which are prefixed `Concrete`, then represent the actual implementations. Concerning our concepts from section 3.5 (p. 117), a `Recommendation` is an element of a result list and a `RecommenderSearchStrategy`, as part of the top-level package `resuse.mrf.strategies`, realizes a sequence of model recommendation operations. These are threaded, because they inherit `Thread`. For differentiation and inspection reasons, each `Recommendation` can be represented by a distinguishing icon as defined by a `UIContributer` and can be previewed. Often, this functionality is needed for `RecommendationUIs` while a result list is revised, e.g., while stepping through a dropdown window that presents the result list under a query box. Note that, in this example, the query box is the actual `RecommendationUI` and is placed in a top-level package called `reuse.mrf.ui`.

The part that is missing deals with the `RecommendationContexts` from the top-level package `reuse.mrf.context`, and we extend figure 4.12 by figure 4.13 [Dol14]. This enables the editing sequences for our context by means of a `ContextHistoryManager`, which is accessible for `RecommenderStategys`. It offers the editing sequences introduced in subsection 3.5.4 (p. 123) and supports the finding of `RelatedElements`. In addition, the filtering of sensor data in tiers, which we mentioned in subsection 3.5.4 (p. 123), takes place here, e.g., for dummy elements.

For a more elaborate understanding of the model recommender framework, it is

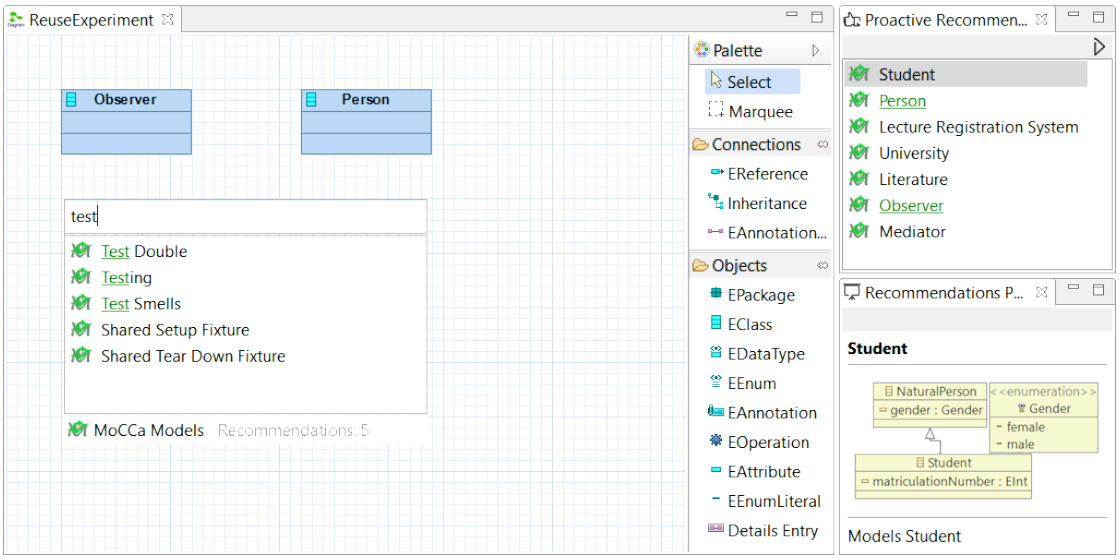


Figure 4.8.: Model Recommender Simulation Environment similar to [DGL14a]

beneficial to glance at object flows, i.e., a sequence diagram [DGL13]. We omit this for the sake of brevity and provide an idea only. A `RecommenderUI` starts a `search()`, and then delegates it to `RecommenderSearch`, which commences the `startSearch()`, initiating all registered `RecommenderSearchStrategys`. In case a produced `Recommendation` is picked, it is `applied()` in a `RecommendationContext`. Note that the actual sequence diagram is slightly more complex because it includes notification, threading, and context determination for picks [DGL13].

Dashboard: Getting started with our framework in new environments or with recommender strategies requires some manual tasks, which we can automate with a dashboard. This includes a whole set of Eclipse plug-ins that must be registered, and provides skeleton source codes that we consider a good starting point. The latter comprise our observations from section 3.5 with respect to operation sequences, schema, and states. Further, it offers guidance and basic configuration options for recurring scenarios. As an example, consider a `RecommenderSearchStrategy` querying a relational database.

Not only can we generate new `RecommenderSearchStrategies`, which are simply called Strategies in our dashboard, but we can also provide starting points for GUIs and context components, as shown in figure 4.7, as well as for so-called launch configurations. They take care of all settings necessary for starting a new environment with all the just-created plug-ins running. This means that a developer simply needs to build the actual algorithm, i.e., model recommendation operation sequence.

Simulation Environment: Simulations can help build a model recommender, because often developing them entails some common difficulties regarding the `Recommendations` and `RecommendationContexts`. These may be difficulties in building well-formed content or in transforming content to an operation-based format that matches the editor dialect.

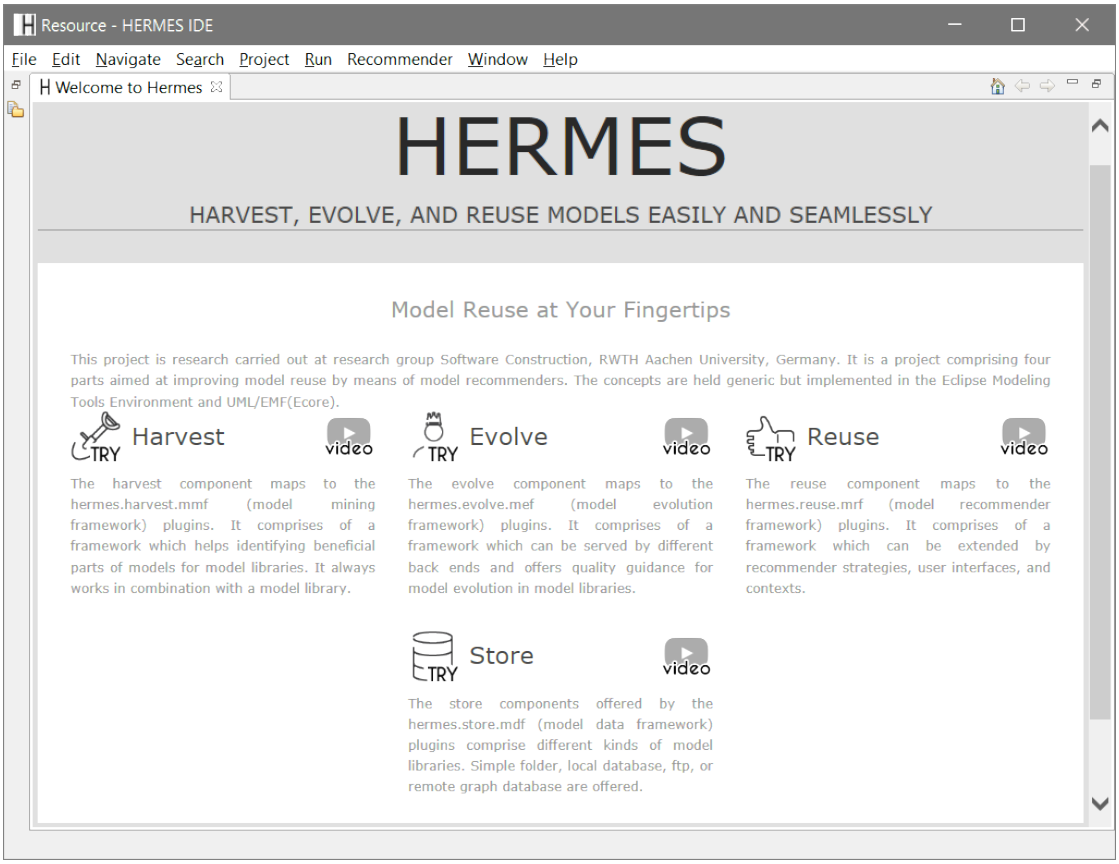


Figure 4.9.: HERMES Welcome

Many issues, which might appear while developing, can be detected through logging methods.

Therefore, the simulation environment hooks into the framework and controls involved elements [Nim13]. This finds erroneous operations that a context produces while applying a `Recommendation` or their irregular content while hiding other framework internals. We build this on the large theory of simulation that is available and define our simulation environment and the simulation model [Ban98; DGL14a]. Without going into too much detail, the simulation cornerstones are as follows: The objects under simulation are `Recommendations` and how they are produced by a `RecommenderSearchStrategy`. The simulation environment is the simulator, as shown in section 4.5, which looks like a graphical class diagram editor. In terms of xUnit test patterns, this simulator works similar to a test spy and records all operations before and after execution [Mes10]. This allows erroneous `Recommendations` and operations to be identified. Finally, the simulation model is Ecore as the least common divider of models that we consider (alternatively EMOF). In addition, the simulation protocols are textual descriptions of attempts to apply recommendations. This categorizes our simulations as activity scanning [Ban98].

Figure 4.10.: HERMES Icon, Symbol, and Overview (cf. figure 1.8 (p. 12) and see figure 4.1)

4.6. HERMES Demo, IDE, SDK, and Design

Careful readers will have noticed that some screenshots above comprise instruction notes in models. For example, these exist for `.store.mdf` (in figure 4.2), for `.harvest.mmf` (in figure 4.3), for `.evolve.mef` (in figure 4.4), and for `.reuse.mrf` (in figure 4.6). They result from demonstration setups in our HERMES prototype, which users can chose for experiments. All that users need to do is choose the respective TRY in the welcome page shown in figure 4.9. Immediately after the setup is complete, the user will face the screenshots mentioned above, and can carry out the steps explained in the note. These are also illustrated in videos with complementary comments for each of these parts [Gan14f], which can be reached by choosing the video on the welcome page (cf. figure 4.9): for Harvest [Gan14b], Evolve [Gan14c], Reuse [Gan14d], and Store [Gan14e]. In addition, a video on the installation process is provided [Gan14a].

Next to these experiments on demonstration data, the HERMES IDE product features the dashboard and simulator, as introduced above [Gan14g]. Hence, we can implement new context components, GUIs, and recommender strategies. However, for further insights, the HERMES SDK is necessary. This ships all the source code, which allows insights to all the other parts, either via debugging or source code review.

As a final note on our realization, note how the design of HERMES is conclusive and coherent, as indicated in figure 4.10. Reading this figure from left to right, the icon is just an H and the symbol reduces the essential parts. These reappear in our project overview, which also functions as our means of structuring concepts. Further, we derive our conceptual and technical architecture from it (cf. figure 4.1). Other than that, the design of the website, welcome page, and project documentation adheres to a guideline [Gan14h]. For example, the welcome page of the HERMES IDE matches the style of the website, with very few adjustments.

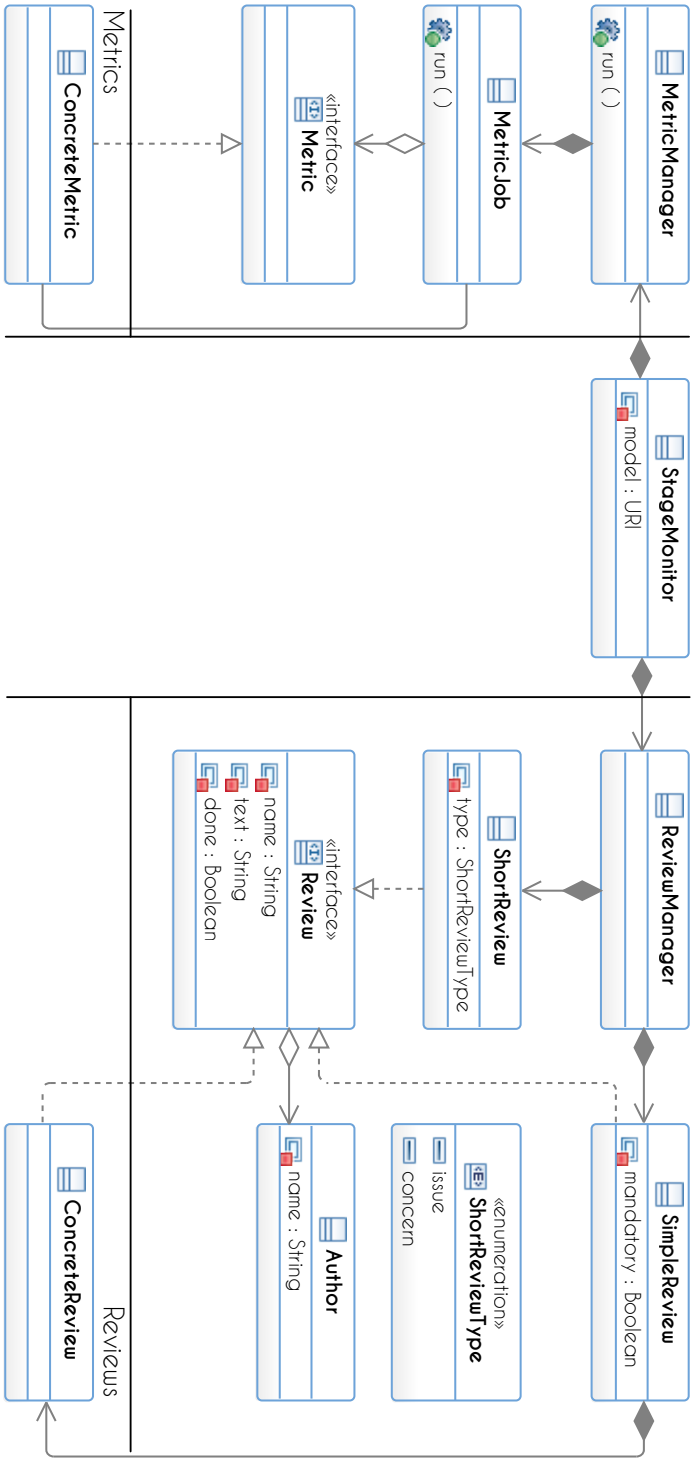


Figure 4.11.: MEF Detailed Architecture

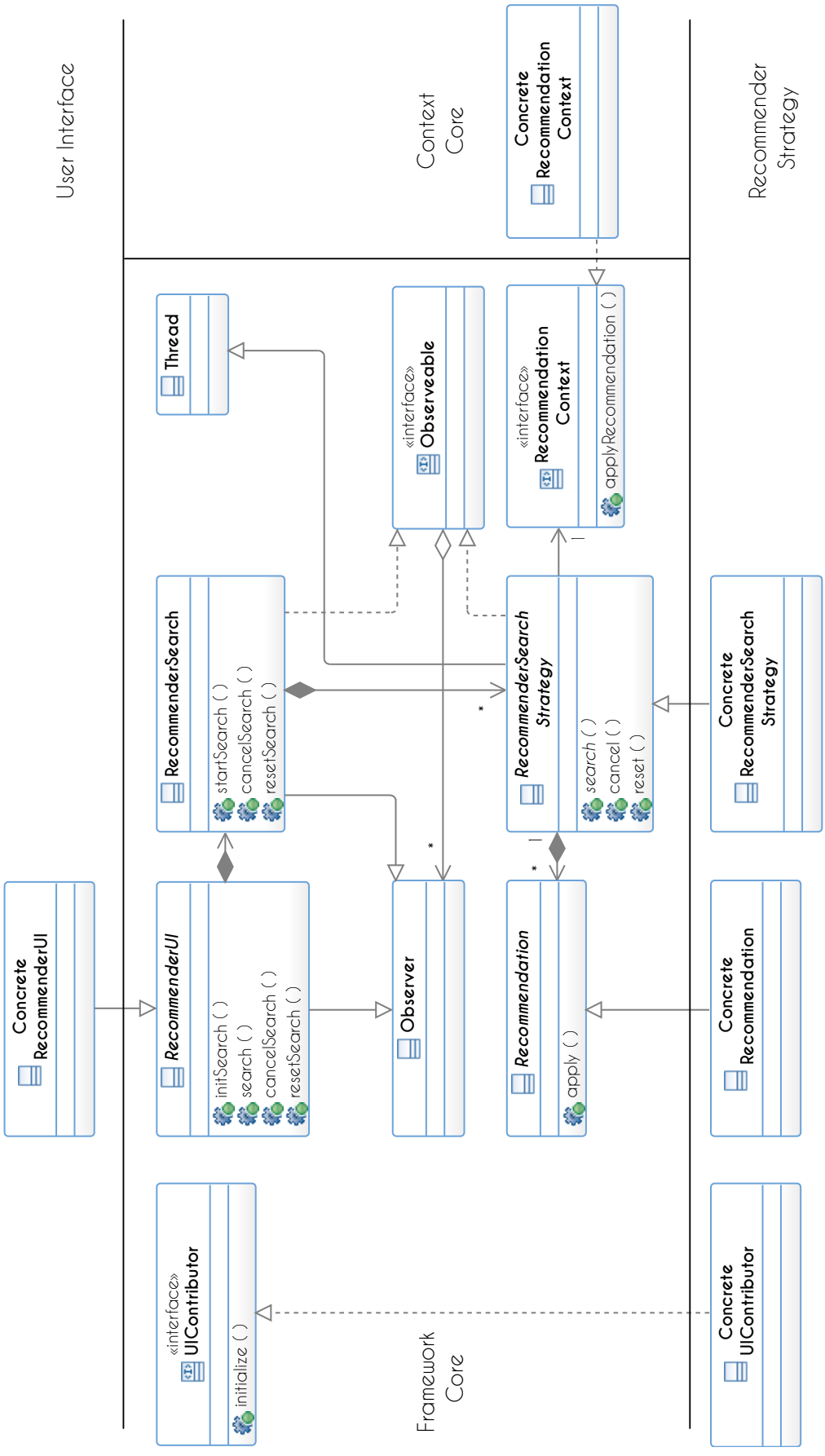


Figure 4.12.: MRF Coarse Grain Architecture (cf. [DGL14a] extended on page 164)

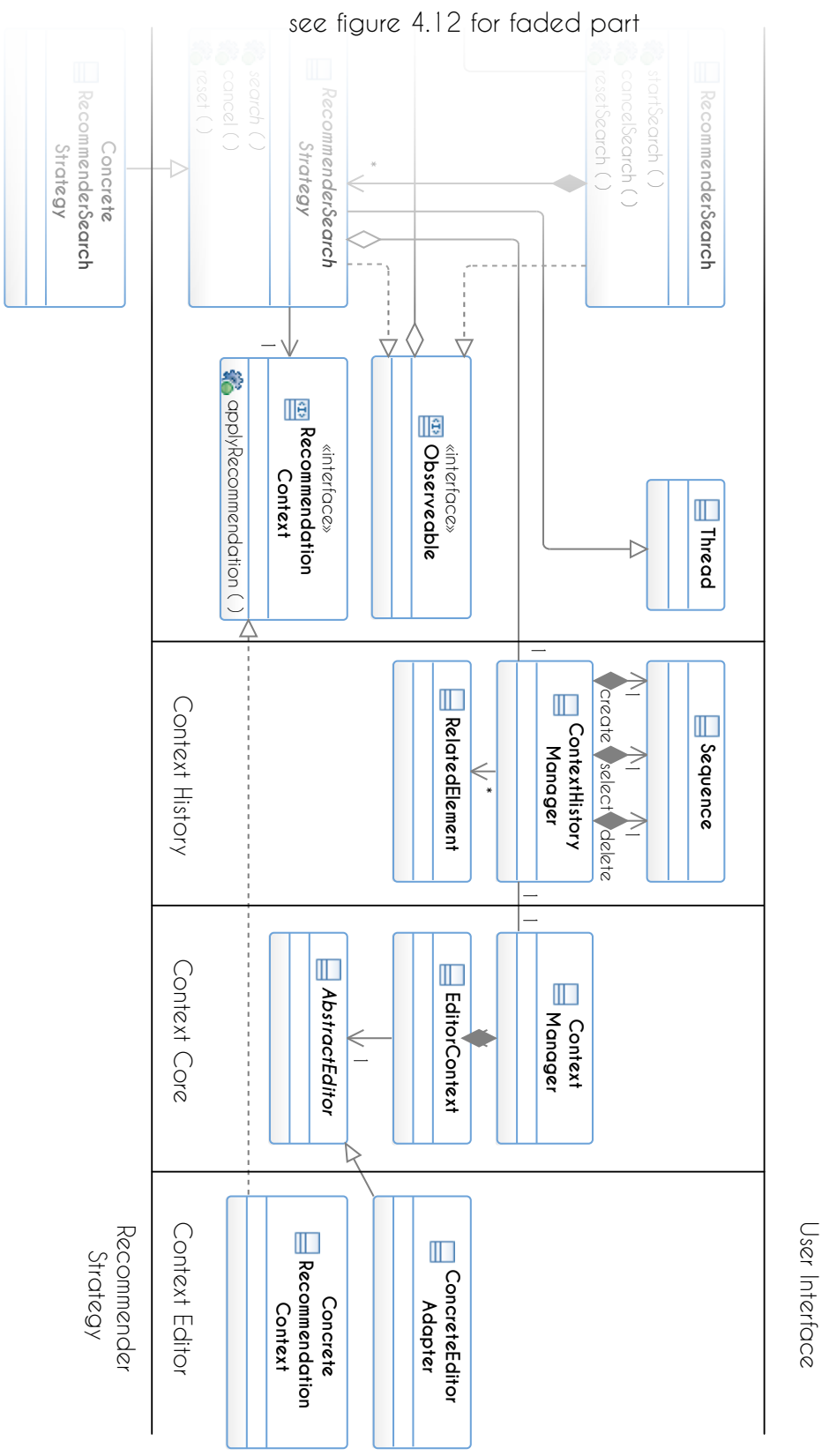


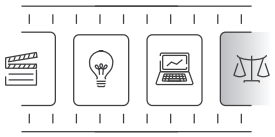
Figure 4.13.: History Extension similar to [Dol14] (complete diagram: page 201)

5 Assessing Processes, Concepts, and HERMES

There are no facts,
only interpretations.
(Es gibt keine Tatsachen,
nur Interpretationen.)

Friedrich Nietzsche

Contents



The approach presented in this text is the result of research conducted in software engineering. Hence, any validation requires an understanding of what "software engineering" actually is [Woh+12b]. The ISO, IEC, and IEEE explain it as "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software" [III10]. Note how this deals with realization within a process, cf., "application of x to <phase>", and how it emphasizes the product characteristics of software, cf., "application of x to software", because everything refers to it. Finally, note the word "quantifiable". We can roughly translate this as "measurable" for our validation purposes. Hence, we can focus on the process and product characteristics, or the process and product quality.

Given this understanding, we can attempt to find means of assessment. Some methods that have become popular in software engineering research concern empirical studies to counter the perceived lack of them in software engineering research [Tic+95; GVR02]. For example, (controlled) experiments, case studies, and surveys became more popular as means of assessments [Woh+12a; Run+12], although they have been questioned as "an academic exercise" [Seg03]. This is because of the one-to-one transfer of these approaches from the social sciences to software engineering research. Hence, "case studies have been criticized for being of less value, being impossible to generalize from, being biased by researchers" [Run+12]. Contrary to that, social sciences and psychology have a large body of knowledge about confounding variables, which are expected to interfere with the controlled variable. For example, the Hawthorne and Rosenthal Effects [RD39; RF63] (Pygmalion Effect [FP79]), whereby subjects behave differently because they are being watched and self-fulfilling prophecies that occur because researchers want a certain outcome, are countered using approaches such as double-blind studies or control groups. We are not aware of reasonable practices for software engineering research that scale studies to the extent that statements can be derived from them. This is not to say that we should not assess software engineering research empirically, but that we need to emphasize the grain of salt we should take the results with!

Hence, our big picture of assessment builds on traditional process and product quality [II11], which can also be considered as approaches based on analytical, constructional, and organizational quality assurance [LL10]. Product quality, in our understanding, is covered by the ISO/IEC standard series 250xx quality models, which enable analytical quality assurance by establishing grounds for analysis and finding issues. Other than that, we consider process quality as constructional and organizational quality assurance and see it as a type of preventive quality assurance. Thus, only together can all these aspects paint an appropriate evaluation picture.

However, the category of software engineering research is another delimiting factor concerning validation. In terms of Glass, Vessey, and Ramesh, our research is conceptual analysis (CA) and implementation (CI) [GVR02], but we provide more than a concept, a proof of concept by means of our implementation, and its measurements. Still, we mostly conceal our initial investigations regarding a solution concerning model reuse as well as UI considerations [Dyc12a; Dyc12b; DGL14b]. Therefore, the guiding challenges that we initially set in section 1.2 (p. 6) are the best grounds for an assessment regarding how much progress we can claim:

- *Storage Challenge:* locating and accessing (Φ, Ψ).
- *Representation Challenge:* warehousing and organizing ($\mathbb{KL}$, $\mathsf{I}^{\epsilon}_{\mathbb{KL}}$).
- *Harvesting Challenge:* identifying and extracting ($\mathfrak{ID}|_{D'}$, $\mathsf{s} \subseteq \mathsf{m}$, $\bigtimes \mathsf{s}_i$).
- *Evolution Challenge:* changing and improving (A_{staged}, $\{-,\circ,+\}$, QG).
- *Retrieval Challenge:* querying and retrieving (MP).

Comparing these pieces of our puzzle to those set as challenges in section 1.2 (p. 6) reveals that each line is extended with the most important symbols representing the solutions, which we have developed, to each challenge at hand. As a brief recap, we constructed an approach for operation-based model recommenders in the previous chapters and explained the implementation we put on top of it. We did so by using middle grounds for the formality of operation-based models and wrote down a complete approach for model reuse, which starts with harvesting, proceeds to quality assured evolution, and terminates with reutilization of models. The demonstration we build by means of a software prototype implements all these concepts and shows how the approach can be brought to life.

Further, despite its conceptual complexity, this prototype can provide a showcase in a video of slightly over three minutes, which covers all basic functionalities. In more detail, given a proper setup, videos for harvesting [Gan14b], evolving [Gan14c], and reusing models [Gan14d] cover each of the use cases introduced in figure 2.1 (p. 25). The exact combined runtime is 3 min and 14 s, which already indicates that we were able to meet two of our main goals: “Integrated Simplicity”, i.e., the “**e**asily and **s**eamlessly” in HERMES.

Otherwise, assessing this research comprises some obstacles, because reliable and significant statements might result from many confounding variables (cf. [Rob09]). First,

developers and modelers prefer to stick to their work habits, and adopting these requires long-term cultural change. This becomes more difficult for reuse, as it inherits a rather inconvenient reputation. Even assuming model reuse in operation, it would be difficult to assess, because it suffers from the same measurement difficulties as have repeatedly been shown for other reuse. Second, there is no knowledge library or model repository on which the research community agrees. Certainly, this is because we are still in the early stages of model reuse recommender system research. Third, any change to a knowledge library interferes with the objectivity and repeatability of results. This could be approached with a commonly agreed knowledge library, but harvesting models once will affect the result.

Subsequently, we will describe the project history to lay the grounds for evaluating our development process, assess the prototype, provide an overall quality discussion, and comment on our contribution to the scientific knowledge base. In more detail, we use our project history to explain why and how we did not get things right at first, and, more importantly, how our development process helped to improve the situation. Further, our discussion on the quality of our prototype shows how we could align the concept, implementation, deployment, and usage experiences for mutual benefit. This is of importance, because each aspect builds on the others in this order.

5.1. Some Project History

Some tasks in software engineering proceed iteratively until they are acceptable. This iterative development was the case for several of our components. Some of them can be plugged into the frameworks we provide, others cannot. Examples of the former are the splitter explained as the clustering algorithm for harvesting in subsection 3.3.2 (p. 75) or the recommender strategies for which we lay the groundwork in subsection 3.5.2 (p. 118). We also opted for this flexibility because we could not see an ultimate solution, but rather grounds for experimentation. However, the main part that was not realized as a framework, although it took several attempts to get a solution, was the knowledge library. For us, the magic number of versions was three, as it was the third attempt at a solution that resulted in a real-world deployment that was conceptually sound. Still, it makes sense to look into all three knowledge libraries briefly.

The first knowledge library was an EMF prototype that helped to elicit the actual requirements we could ask of a knowledge library [Boh11]. We did not name this prototype, but designed it in such a way that it could be used for code generation. The resulting prototype served as grounds for experimenting with the meta-structure that we eventually published [GL13]. This was particularly useful, as the EMF generated (the so-called tree editor) comes with the option for dynamic instances. These require no generated source code, but can use the reflective API. We learned from this prototype that we should expect the meta-structure in the form of a graph and that access to this structure and the content it represents, e.g., models, must be simple. Hence, we experimented with several third-party EMF add-ons for forms, querying, and indexing models. Note

that this prototype realized an approach similar to mega-modeling [Béz+05], i.e., one model as a meta-structure for references to other models. The differences were the links between models, which we provided, and the cross-project nature of our meta-structure.

The positives of this prototype were its simple and rapid development and seamless integration into Eclipse. The drawbacks lay in distribution support, which also encountered a lost update syndrome because persisting data under version control permitted no transaction support. In addition, we faced severe issues with querying, which were only recently considered by EMF IncQuery [Ujh+15]. With these concerns in mind, we explored the available technologies.

The most complete package as a data provider for our meta-structure and its content appeared to be a graph database [RWE13]. Hence, this led to our second prototype, which we call MoCCa ("Model Composition and Combination Vault", `.mdf.mocca`) [Fuc11]. The name already indicates that only treasured models are meant to be persisted and that composing or combining these models is a vital requirement. Additionally, we experimented with several other ideas for organizing models and the functionality grew quickly. We even started experiments on harvesting, splitting, and merging models, which are all indirectly supported by MoCCa.

Thus, we managed to fit all functionality into one system by integrating different technologies. A graph database represented the meta-structure, a version control system managed the actual persistence, and an indexing and querying system fostered sophisticated retrieval. Once again, we could build this system quickly, because Neo4j, Lucene, and git operate well together and we achieved separation of concerns regarding technologies. One drawback was that we had to come up with our own UI to fit in with Eclipse, and another issue was the hardcoded nature of MoCCa. This made it difficult to extend, e.g., for demonstrating model evolution. Altogether, we found that MoCCa suffered from "second system effect", which states that second versions are often overambitious and try to fix everything that was wrong with the first version [Bro75]. Hence, we either had to chop down our solution or turn it over again. After reevaluating the solutions available on the market, we started over.

The final realization, which is simply denoted as `.mdf.store`, was the result of combining the two former prototypes while meeting additional requirements [Tra13]. We took our experiences from the generated prototype and fitted a more sophisticated graph database as a persistence strategy, which is called the resource strategy in EMF. As an alternative persistence, we built in support for both graph databases and relational databases. We did so because deployment scenarios in industrial contexts might not be willing to install another database, so our solution had to fit into existing environments, and we are aware that many relational databases have been deployed. In addition, we built our indexing more flexibly than in MoCCa and implemented a registry for it. This is why we talked about extendable indexing and querying in subsection 3.2.2 (p. 56). Further, we distributed responsibilities to several servers for the meta-structure, preserving the actual models, and indexing/querying. Overall, we gained a distributed and scalable solution that provides the functionality needed for model recommendation production, and we shared this with the model repository community [TGL13]. As an extra experiment, we

made MoCCa available under the same API.

Altogether, provided proper installation, we came up with a knowledge library that is flexible enough for extensions, scales well, operates extremely quickly, and is designed for professional deployment. The downside of this solution is its high entrance barrier and rather complex setup, because the technologies we employ, despite their performance, come at that cost. Still, we were willing to pay that price to enable responsive model recommendation production [DGL14b; DGL14a].

Regarding the development of harvesting and reuse, we built and leveraged a framework architecture for each case. In other words, even three iterations were not enough to get them right—and we are not certain that there is an ultimate truth for clustering, splitting, or model recommendation production.

5.2. HERMES Quality and Experiences

Quality is known to be a complex matter, and we started this chapter by distinguishing between analytical, constructional, and organizational quality assurance [LL10]. We will cover these subsequently, but first provide a more detailed understanding. We put analytical quality assurance, measured by means of ISO/IEC 25010 quality models, under product quality [II11], and we further make the distinction between internal and quality in use aspects with all their sub-items. Further, we elaborate on our means of constructional quality assurance by discussing our process quality. This also includes organizational aspects.

Other than that, one train of thought needs further clarification: The quality of our software prototype is based on the foundation, which we provided through our conceptual groundwork in chapter 3. For example, our approach builds on operation-based models and knowledge libraries, i.e., both together facilitate the parts concerning harvesting, evolution, and reuse. In other words, they establish a language we could use to explain harvesting, evolution, and reuse. Therefore, we can consider them as evaluative parts, because they allowed us to reach a functional stability. This is because of our knowledge that operation-based models and knowledge libraries were complete, correct and appropriate for formulating harvesting, evolution, and reuse of models. However, research is often not quite as straightforward and rarely goes as planned. Hence, we start with a few remarks on our concept, although iterative development enforced continuous feedback loops between software and concept.

5.2.1. Process Quality

As much as we could benefit from iterative development for our concepts, “[f]or a process, it is not possible to build a prototype” [RRS11]. Even more, we agree that “we will never find the philosopher’s stone” in respect of a rational design process [PC85]. Hence, we “faked it”, as discussed in chapter 3, by concealing quite a few of the twists and turns we took. In other words, “[w]e [did] not show the way things actually happened, we

show[ed] the way we wish they had happened and the way things are" [PC85]. This makes it feasible to replicate our results. Still, there was a design and development process and we applied best practices for scientific software projects [HLN09]. These best practices and their associated development process for scientific software applied in the HERMES project mainly adhered to the results of our working group [Hof13; HLN09], as summarized in figure 5.1. Certainly, some alterations were required because of technological changes and the individual requirements of our project. Without going into too much detail, we present our process.

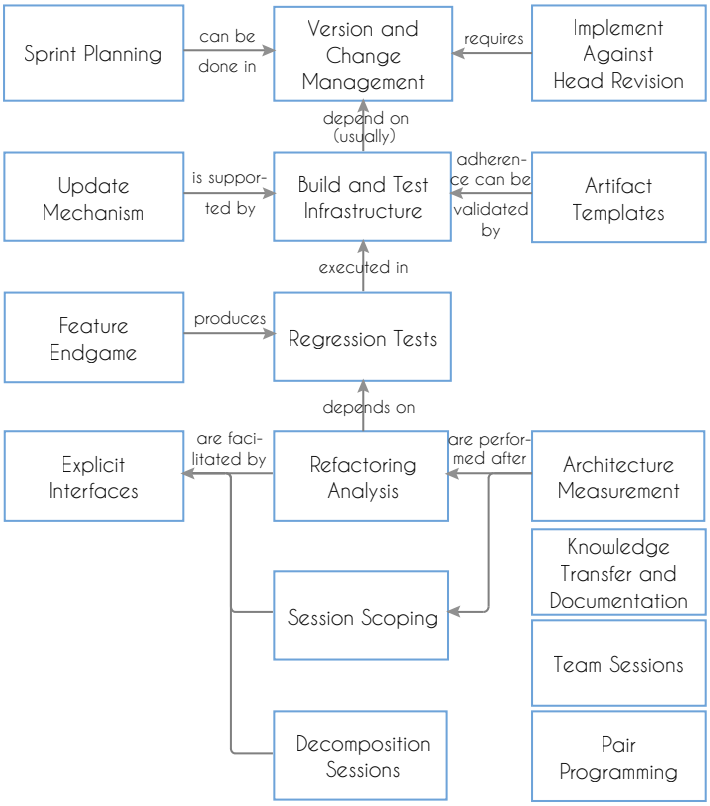


Figure 5.1.: Best Practices for Developing Scientific Software similar to [HLN09]

First, readers will have noticed that we were very serious about related work. Not only is this due to a strong will to cover related work, but also to gain as much as possible from the knowledge base, thus stimulating input for discussions. Additionally, we were always reluctant to introduce new symbols, terms, or notations, but could rarely base our work on common agreement. Instead, we often had to pick the most common usage. Overall, an approach we often applied for systematic literature review is very similar to a proposal by Wohlin et al. [Woh+12c]. Hence, we defined scopes for need, focus, and technology, but not with respect to experimental designs and outcomes. The latter is attributable to a lack of availability. In addition to the scopes, we exploited literature database functionality for alerts in two respects. The first was keyword alerts, which

automatically notified us about these keywords in new publications, and the second was citation alerts for particularly important publications. These notified us about citations of this work in other publications, which may be completely unrelated.

With an agreement set for terminology and understanding derived from related work, we set to incremental development as proposed. This means we had weekly team meetings for planning, nightly builds on a Hudson server applying findbugs, PMD, checkstyles checks, code metrics, and Junit tests. Further, we had weekly integration cycles on a base of Eclipse features and plug-ins. This is all identical to our colleagues' findings except for the complexity of our platform. This was reasonably smaller in our case, because we could make use of a more recently provided Eclipse architecture functionality.

Each developer had an Eclipse product, our HERMES SDK, provided for development, which had the benefit of making every developer a user at the same time. Sometimes, this is referred to as "eat your own dog food". The HERMES SDK was always bundled with the same checkstyle, PMD, findbugs, git, Trac, and SonarQube configuration as those applied during nightly builds. Eventually, SonarQube took over the checks of interest, so we could solely use them.

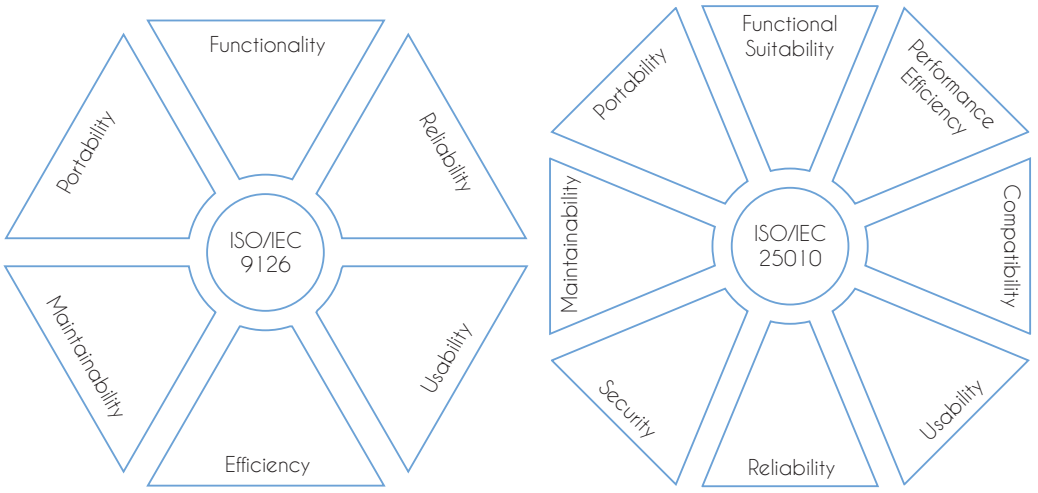
The build infrastructure that we introduced followed a one-click setup, one-click run approach. This means that all configuration necessary for a local build by a developer could be installed with one script, and the build could be run by leveraging another script. This is important, because our infrastructure grew over time. Eventually, it comprised a Maven/Tycho build relying on a Hudson build server, Neo4J graph database, Rexster graph database server, MySQL database server, elasticsearch indexing and search server, SonarQube measurement server, and git version control server. We did this heavy lifting in respect of infrastructure because we desired a prototype that runs well, although we never aimed for a commercial-quality product. This is different to our colleagues' approach to run mostly a central build server, but they could not set up an infrastructure as easily as with Docker containers. Further, Maven/Tycho was not available at that point, but eased our build infrastructure enormously. As a bonus, which we considered only a nice add-on but eventually used a lot, there were so called long-term builds that enabled us to deploy based on different versions of Eclipse. This eased migration to the most recent version of Eclipse.

Further, we should discuss the differences between our development, which resulted from git and handling by git-flow. Our colleagues only discuss version control and change management for cvs and svn styles, but we applied git, which is a distributed version control system. This brings about the possibility of development by means of git-flow, which allows for branches for every feature developed. Hence, developers were almost independent and only advised to "rebase" their development on a regular, i.e., weekly, basis. This means that their branch was continuously rooted on a new "Head", which is called "Develop" in git-flow. Overall, this eased merging and migration considerably. Another benefit of applying git-flow resulted from public branches for features. These enabled developers to create Trac tasks with Eclipse Mylyn contexts attached whenever they were struggling with something. These Trac tasks could be checked by any team member, with all the changes highlighted in the files added to the Mylyn context. With

additional discussions and an escalation mechanism, this enabled us to accelerate and distribute development fostered by change management using Trac.

5.2.2. Product Quality

Measuring software engineering products is often bound to measures, (code-) metrics, and quality models. For the purpose of this assessment, we talk about measures and not metrics, because we do not introduce a detailed measurement theory [Via16]. All we need to know is that metrics are usually considered measures with an interpretation. Further, quality models subsume and arrange aspects of consideration for assessment. We employ two examples of quality models, as depicted in figure 5.2. One is the current quality model provided by the ISO/IEC 250xx SQuaRE series (cf. figure 5.2b) [II11]. This replaces ISO/IEC 9126 from 2001 (cf. figure 5.2a) and renders it obsolete [II01]. Regardless, we provide this for comparison and because it is still very popular for assessments. However, we omit any further discussion and leave the comparison to readers. Subsequently, we guide our assessment using the structure provided by figure 5.2b, moving clockwise for each of our components and linking them occasionally to our product metrics, as provided in figure 5.3. Note that we only introduce the top characteristics and not every sub-item. Instead, we simply use the latter. Further, we do not take into account data quality (cf. ISO/IEC 25012).



(a) ISO 9126 Product Quality (b) ISO 25010 SQuaRE

Figure 5.2.: Software Engineering – Product Quality Models [II01; II11]

Functionality Suitability is the “degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions” [II11]. First, our prototype meets *functional completeness* on a coarse level because it realizes the use cases we introduced in subsection 2.4.1 (p. 24). We can state so because it was the focus of iterative and incremental development, and we can provide a showcase

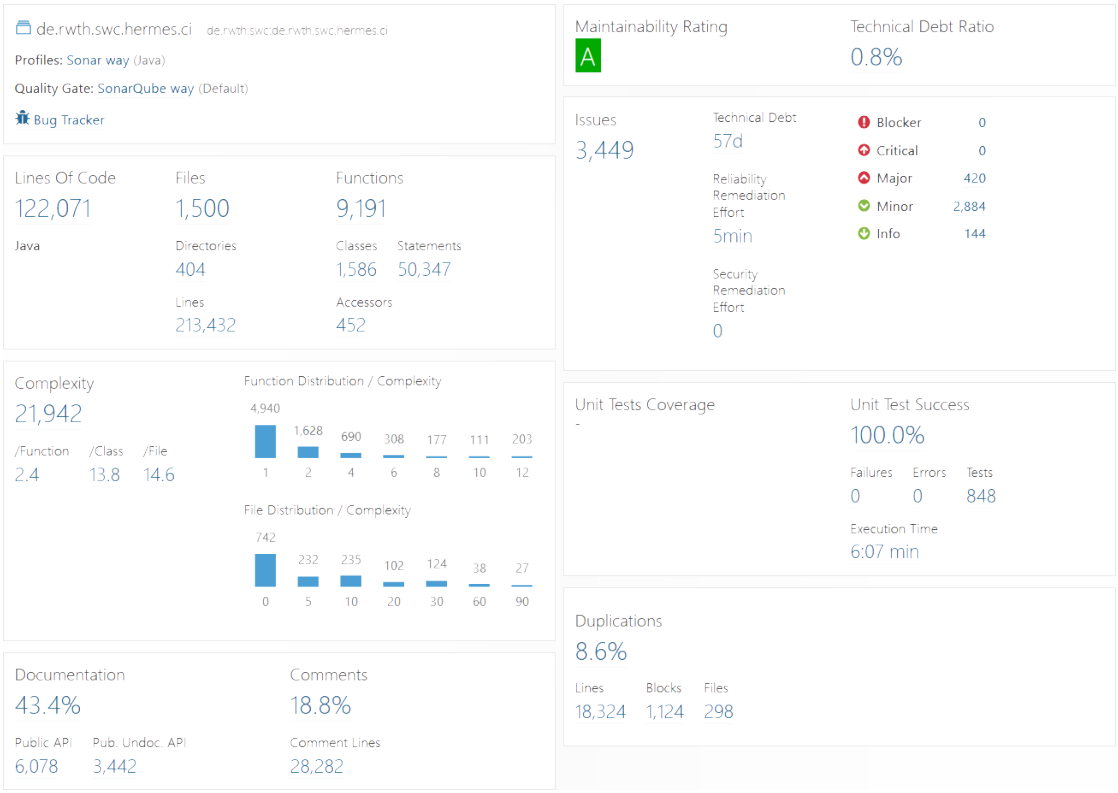


Figure 5.3.: HERMES 1.9.1 SonarQube™ 5.6.1 LTS Measures

slightly longer than the three-minute video mentioned above. On a more fine grained level, our prototype is functionally complete to a certain extent, because our framework is suitable for extensions. Thus, despite our demonstration of these extensions in several ways, others have not been provided. For example, we provide several means for clustering models, but a semantically reliable clustering is not among them. One might state that this was not the primary goal of this research, which is certainly correct, but it does not change the fact that this is a reasonable user objective and we only made some progress in respect of automating the harvesting process. Second, our prototype was meant to be and to remain prototypical, which already places limitations on any possible *functional correctness*. Hence, the 850-odd tests that helped in development and ensuring functional stability only provide a weak statement. Still, they assured us in moments of alteration that functionality remained. Other than that, our SonarQube server, PMD, and findbugs (cf. figure 5.3) allow a certain statement of correctness, and we believe that reasonable performance was attained; after all, quality is relative. Third, our prototype, in our understanding, performs well in regard of *functional appropriateness*. We believe so because we reached a state that overcame the second system syndrome discussed in section 5.1 in many ways, not only regarding our knowledge library.

Turning to the approach introduced in chapter 3, our assessment of functional suitability

is slightly different. First, *functional completeness* is good, as we can derive information from the numerous examples, as shown in the course of chapter 3. However, we lack *functional correctness* in terms of precision, because we needed to cut things short for the sake of comprehensibility and brevity. Otherwise, a fully formalized concept would take several dozen more pages without providing additional findings. As an example, we used a mechanism known from object orientation for building the sets a model is made of in our approach, so we can make more detailed statements about elements and relationships. We even introduced a typing function without defining it in great detail. Instead, we used this understanding and provided pointers to others who elaborate on what we omitted. Third, *functional appropriateness* appears good to our mind, but some small points could be considered over-engineered. This is because, on the one hand, there are elaborate possibilities for making use of the operators we introduced, e.g., for model recommendation production. On the other hand, this is because of some functionality that is not obviously used in our approach. The simplest example is our revert operation for models, but there are several others of this kind. Still, we retained them in our approach to keep it sound. In the example mentioned, this means an attempt to show that we implement a Boolean algebra remains possible (cf. [Kög11]).

On a more general note, we did not explicitly discuss some common issues of recommender systems. First, this regards the cold start problem, i.e., what should a recommender do if there is no or very little information available to take into account for model recommendation production. In terms of Amazon, this would be a new user with an empty profile, and in our terms this would be an empty context. Second, we presented a dual-mixed hybrid recommender approach in figure 3.36 (p. 141) and discussed merely more than knowledge-based recommendation system approaches. However, there are many other recommender approaches available. It also means that we omitted a discussion on very simple recommender systems, which use features simulated by our indexes and transfer basic ideas of recommender systems. Given our views on models and editing sequences, it is trivial to use a common recommender system library. Third, we omitted discussions on malicious use or other attempts to break our approach. We did so because, in the current state, they seem negligible, though they are possible. Ideas on that are provided for regular recommender system and might also work for our approach [Pic+11; BR11].

Performance Efficiency is the “performance relative to the amount of resources used under stated conditions” [II11]. First, we did not set *time behavior* constraints for our prototype, which implies that its prototypical nature offers potential for improvement. While this is very true for our HERMES IDE and HERMES SDK, the final knowledge library introduced in section 5.1 is built to respond quickly. This is why we allow for distribution to underlying servers. Other parts critical to time behavior are clustering and model recommendation production. In our prototype, we chose the reasonable way, meaning that we employed caching whenever reasonable, but did not reinvent the wheel for clustering. Second, *resource utilization* is a general issue with Eclipse products. They are often considered heavyweight, and this is also true for our HERMES IDE and HERMES SDK product, but compared to a pure Eclipse, we do not impose additional issues, i.e., we

add no significant impact on its start behavior or general responsiveness. Third, *capacity* is only an issue for our knowledge library, and we consider distribution reasonable for scaling. Regarding our approach from chapter 3, there are no aspects to discuss.

Compatibility is the "degree to which a product, system, or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment" [II11]. While this is not an issue for our approach in chapter 3, our prototype, despite its size, works well in terms of *co-existence*. This is attributable to, firstly, the products we provide, if we think of effects on the system other than the product itself. Second, this is attributable to OSGi and the architecture that we developed, because they provide the means for encapsulation that we frequently employed. Third, the *interoperability* of our prototype was a minor concern, but data and communication in our prototype adhere to (de facto) standards. For example, models are based on Ecore, persistence by means of XMI, and communication with elasticsearch, and the Rexster server is based on JSON.

Usability is the "degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction in a specified context of use" [II11]. Next to the requirement to obtain a basic solution, the top priority is stated in the project name: "easily and seamlessly". We can demonstrate a basic showcase of model reuse as a process by means of the HERMES IDE in five videos of approximately 60 s each (total 5:07 min). They illustrate the core features including downloading, installing, and experiments for each part from our domain architecture. If we are limited to the three core use cases, the videos last 3:14 min in total. Note that this is all despite our rather complex foundation. Other than that, the UIs were continuously evaluated in "desktop surveys" and through a final assessment by "laboratory experiments" [Woh+12a], e.g., [Sew13; Rot12; DGL14b; Dol14].

With regard to usability sub-characteristics, first, *recognizing appropriateness* was no concern for our project, because users need to know if model recommendations are appropriate for their needs in the first place. Second, we consider *learnability* as a positive aspect of our approach, because it does not expose many concepts to the user despite their need for the approach. Still, reusing models is certainly simpler than harvesting or evolving them. In particular, the latter comes with quite a few new concepts, e.g., stages, simple reviews, or review hats, which are not always intuitive. This is why we provide a lot of user guidance in our software prototype. Third, *operability* is influenced by what we mentioned before, because user guidance enhances operability. Other than that, we consider a quick learning video for harvesting and reusing models to be more beneficial than elaborate tutorials or handbooks, because in either case a few keystrokes will expose the necessary functionality. Still, the setup of a deployed system requires some effort. Fourth, *user error protection* was of no concern for our prototype, so system behavior might suffer. Fifth, *UI aesthetics* are a positive aspect of our prototype for two reasons. First, many constraints are posed by the development of Eclipse, so we have little ability to change the aesthetics, and second, we were concerned about a coherent design for our project and product, as stated in section 4.6 (p. 161). Finally, *accessibility* was not a real concern, because we are covering a niche market.

Note that usability concerns do not entirely apply to our approach in chapter 3. We only ensured that commonly agreed notations and symbols were employed as much as possible, and only introduced new symbols if necessary. Hence, we hope that we achieved a notation with easy learnability, which we foster in our glossary and index.

Reliability is the "degree to which a system, product, or component performs specified functions under specified conditions for a specified period of time" [II11]. First, *maturity* was only a minor concern for our prototype, but we achieved several stable releases over time and the core functionality rarely needed to be changed. At the time of writing, we have reached version 1.9.1 for HERMES IDE and SDK according to Eclipse standard versioning, which starts at 1.0.0. Certainly, this does not mean that we have left the prototypical state, but that we have proceeded further than simply a proof of concept, as we indicated early in this chapter. This has a certain impact on, second, *availability*, because a prototype is often developed following the head branch, as explained in subsection 5.2.1. Because of this, a nightly build could put the system out of service and, third, *fault tolerance* was not a concern. Therefore, users needed to expect some downtime. The same holds true for *recoverability*.

For our approach, it is important that we achieved a high maturity, because we used our notation for some time while writing down and discussing recommender strategies. Some of that is leveraged by our dashboard and simulator. Further, we reached a level of abstraction in our approach that provides the means to use our notation as a language and formulate higher-level solutions. For example, our recommendation operations or schema for model recommendation production and candidate generation are such higher-level solutions (cf. subsection 3.5.7 (p. 138)). Similarly beneficial are indexing and querying operations or our disambiguation for recommender strategy data sources.

Security is the "degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization" [II11]. Research prototypes in software engineering often suffer from security issues unless this is managed, and our prototype is no exception. This means, at best, we delegated security issues to respective services, i.e., our underlying Rexster, elasticsearch, and git server. Certainly, this puts our security in a very good place, but this is only because of the decision to use these servers and not because we considered it in detail. However, next to authentication with these servers, authorization is a real issue in our prototype, because anyone can take any role that is crucial, e.g., for our review mechanism. In other words, any modeler can override issues and put a model in the `fine` stage, although this should only be possible for a limited group of, say, principal modelers. Regarding our approach from chapter 3, we do not see security as a big issue, but we are aware of privacy issues. These are beyond the present scope, but are addressed in our concluding remarks.

Maintainability is the "degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers" [II11]. Disregarding fundamental changes, our prototype exposes certain characteristics regarding maintainability that rely on figure 5.3 and figure 5.4. First, *modularity* is provided at a low level due to roughly 120k LOC (cf. figure 5.3) distributed over about 100 plug-ins (cf. figure 5.4) in OSGi bundles

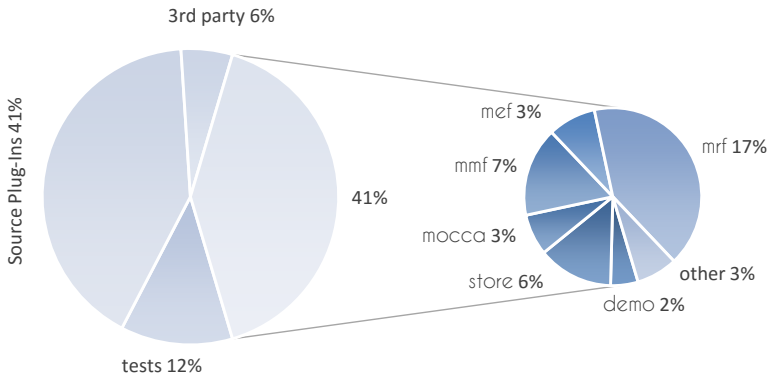


Figure 5.4.: HERMES Plug-Ins Overview n=196

with strict access policies and APIs-exposing functionality. The number of plug-ins might appear high, but the framework character of harvesting and reuse (.mmf and .mrf) allow for plugging in new solutions as needed, which we made use of frequently. On a slightly higher level, the components derived from our domain architecture, e.g., `.harvest.mmf` or `reuse.mrf`, in combination with their EMF, limit further impact on change propagation. Still, *reusability* is low because each component is highly tailored to our domain. However, *analyzability* is high because of the plug-ins we introduced and our logging framework, simulation environment, and the template character of our recommendation operations. Further, *modifiability* is backed by adherence to a style guide and partly fostered by our tests, which support refactoring, i.e., assure functional durability. Finally, we cannot give an overall statement on *testability*, because design decisions were not always equally clever. Some parts are realized for dependency injection, others are hardwired. While the former certainly fosters testability, the latter is often an obstacle to it.

Note that the numbers in figure 5.4 are meant to provide an overview rather than being exact, because next to our HERMES IDE and HERMES SDK exist other products that sometimes feed components back. Hence, the exact Eclipse products vary from time to time. The baseline for figure 5.4 at the time of writing was 196 plug-ins available in our HERMES project repository. Some 160 of these were published grouped in 46 features on our P2 update site [Gan14g]. The plug-ins, which we did not ship, are of test, demonstration, legacy, or branding nature.

The maintainability of our approach appears reasonable, because the operators remain the same and we designed the whole approach flexibly. For example, alternative models, even reduced to vertices and edges, induce only obvious subsequent alterations, i.e., use the element and relationship sets as they were introduced. The same holds true for alterations to our knowledge library in case of a new `LibraryElement`, which needs new indexing and could require new querying to be plugged in. Even clustering in harvesting and cross-links suffer mild alterations; they are based on graph structures after all. Eventually, model recommendation production will need only slight alterations, because our explanations were rather high level anyway. The picture is different for the evolution part. This could change entirely, say, if we put snippets of DSLs in our

knowledge library and need to subject them to quality assurance. This is because a quality model and respective measures are likely to be different for DSLs.

Portability is the "degree of effectiveness and efficiency with which a system, product, or component can be transferred from one hardware, software, or other operational or usage environment to another" [ISI11]. First, *adaptability* did not play a big role in our work because Eclipse, as a platform or framework, provides stable grounds and even supports legacy Eclipse 3.x realizations through a compatibility layer. Further, the servers we use, e.g., git, are expected to be long-term systems and were chosen because of that. Second, *installability* is considered because we found our realization on well-established servers and Eclipse. Regarding the latter, this means we provide packaged products for download that only need extraction, or an available Eclipse can install our components via a P2 update site [Gan14g], granted a suitable Eclipse. Third, *replaceability* is not an issue because we are not aware of a functionally similar system we could need to replace. There are model repositories, but we would rather exploit than migrate them.

Other than that, certain parts that we rely on in Eclipse might change. For example, a new class diagram editor could replace the current version. However, this would require only a changed context, because we use platform-independent operations associated with the context. Similarly, we designed our prototype with the intention of minimizing change propagation. Within our prototype, we could picture the following change: Say we were to introduce another type of `LibraryElement`. This is quickly introduced in our knowledge library through code generation, but it results in some further changes, e.g., additional indexing and querying. However, treating models as sets or operation sequences remains roughly the same, although a new context is required to handle the newly introduced `LibraryElement` properly, i.e., transform them into operations that can be applied to a canvas. Contrary to that, the whole evolution we have introduced is rather tailored to our `Model`, i.e., class diagrams. Hence, we do not consider it as especially portable in regard of our quality model and respective metrics. The stages, simple reviews, review hats, and classification of issues and concerns remain valid.

5.2.3. Quality in Use

Contrary to the mostly internal view of software engineering products discussed above, another perspective is from the outside, or in other words, quality in use [ISI11]. In some respects, this perspective mirrors the evaluation approaches mentioned at the start of this chapter as empirical studies. Hence, there are only a few comments on our approach from chapter 3. Note that we focus on the reuse component and model recommendation production here, because they are the eventual goal.

Effectiveness refers to the "accuracy and completeness with which users achieve specified goals" [ISI11]. The effort required to gain an *accurate* and *complete* solution, given an ideal model recommendation production, depends largely on two factors: the input and knowledge library. Further, the benefit depends on certain known elements. Hence, we could sketch these factors as a mapping, showing that more precise inputs and more elaborate knowledge libraries lead to accurate and complete results, but cost

some degree of effort to define. Further, perfect completeness is rather unlikely, because modeling inherits solution specifics and we aimed at so-called eighty percent solutions.
Efficiency refers to the "resources expended in relation to the accuracy and completeness with which users achieve goals" [II11]. Reused models supposedly need fewer interactions with an IDE and have higher quality. Hence, finding and establishing the solution from a reused model offer some supposed savings. Given models of approximately nine classes, some relationships, attributes, and operations, an assessment of the actions a modeler needs to undertake to create them, i.e., the length of an editing sequence, should exceed the actions needed to find, pick, and alter a model for reuse. This should also be true for our pursued eighty percent solutions. This means that any measure for human–computer interaction based on Fitts' Law will expose benefits for model reuse [AZ03]. Note that this does not take into account the interactions needed to harvest and evolve models. However, repository projects often lead the argument for standardization as an additional higher reuse goal [Som11].
Satisfaction is the "degree to which user needs are satisfied when a product or system is used in a specified context of use" [II11]. Disregarding the quality of data, which we cannot take into account here, we claim that the entire approach and its implementation is *useful*. Not only is this because of the repeatedly stated need for reuse on a conceptual level, but also because of our deployment experiences. We elaborate on them in more detail in subsection 5.2.4, particularly because the recommender strategies, which we do not consider real recommender systems as they do not introduce novelty or surprise, play a major role in getting our approach used. Still, they establish *trust* in our implementation and lower the entrance barrier. Note that personalization, i.e., editing sequences, plays a role here as well. Next, *pleasure* is hard to assess because it depends largely on the data a knowledge library can work with and on the tuning of the recommendation operations. Finally, we consider *comfort* as a positive aspect, because we put model reuse at each users' fingertips seamlessly (cf. S in HERMES) in both a proactive and reactive manner. In the former, we realized presentation in the most unobtrusive way we could find, and in the latter case, we employed well-known user interaction patterns for IDEs, i.e., key strokes or selections.
Freedom from risk is the "degree to which a product or system mitigates the potential risk to economic status, human life, health, or the environment" [II11]. Hence, it should be of little surprise that we did not cover this.
Context coverage is the "degree to which a product or system can be used with effectiveness, efficiency, freedom from risk and satisfaction in both specified contexts of use and in contexts beyond those initially explicitly identified" [II11]. Note that this meaning of context differs from our understanding for the recommender framework. Here, it is meant as an environmental term, so it does not apply to us.

5.2.4. Deployment Experiences

Certainly, the recommender strategies and the data they can build on are at the heart of our deployments, and we gained some surprising experiences in putting HERMES in use.

These findings are mostly work-culture-related and do not concern the potential benefits gained from reusing models. Hence, we introduce some of our recommender strategies and their respective data sources. Note that we use the term recommender strategy, although they might not give recommendations in a sense of RSSE, i.e., they might not follow the entire model recommendation production sequence or the schema introduced in subsection 3.5.7 (p. 138). For example, they might not perform any ranking or filtering. Still, they helped us to derive the model recommendation production schema.

The first recommender strategy is very basic, and we call it QuickText. This provides a means to speed up element creation by leveraging some model operations from subsection 3.1.2 (p. 37). Therefore, it plugs into our model recommender framework by using textual inputs from our reactive query box (cf. figure 4.6 (p. 157)) to create elements on a given modeling canvas. The DSL for processing is very fundamental and we explain it through a few examples: “new Airport, Passenger, Vehicle” will create three classes with according names. “new Vehicle < Bus, Car, Train” yields a superclass `Vehicle` with three subclasses, as expected; this is also allowed to have several superclasses for multi-inheritance. “new Airport - Checkin, Gate, Tower” creates an `Airport` class associated with three classes. In addition, simple comma-separated lists result in respective elements and asterisks between terms represent composition relationships. As a gimmick, QuickText also takes already-present elements into account. This is useful for associating two not yet related but already created elements. Note that elements are mostly classes for QuickText.

Another recommender strategy, which is relatively fundamental, can be considered as an accelerated copy’n’paste reuse, and we call it QuickPaste. This uses folders as data sources and searches them for Ecore models related to the given query. Again, this recommender strategy plugs right into our model recommender framework with our reactive query box (cf. figure 4.6 (p. 157)). The query itself can be a regular expression in Java syntax, and matching models are listed as usual. If picked, the according model is simply inserted into the canvas, hence the thinking that this is an accelerated copy’n’paste reuse. Note that a folder can be located on a computer running an Eclipse IDE, on a shared folder in a network, or in a folder under version control. The latter has built in support from any recent Eclipse IDE. Note that there is an enhanced version of QuickPaste called QuickModel that uses Internet search engines to find models. For example, a Google search of a given query need only be suffixed by “filetype:ecore” to result in Ecore models only. Further, optional parameters such as “site:eclipse.org” or “inurl:github”, provide a means of filtering results that are hosted at eclipse.org or have github somewhere in their URL. With simple adjustments, this brings Ecore models hosted somewhere in the Internet and indexed by a search engine to our fingertips. However, this raises copyright issues that we do not discuss here.

A more semantic recommender strategy, which is also not a real recommender (RSSE), uses WordWeb Online as a database. Given a query, this strategy queries WordWeb Online and provides its results. These can be terms that are related as synonyms, are derived forms of the query, or occur in part of relationships to it. Moreover, these results preserve relationships, e.g., “Male” with a picked recommendation “type of Person”

establishes both classes, with the latter being the abstract superclass. Note that WordWeb Online is vital for model recommendation production with the help of synonyms.

Finally, the MoCCa recommender strategy leverages almost the full extent of our presented approach and makes use of the respective knowledge library. Its content comprises design patterns [Gam+95], some example models, and additional handpicked models from various sources. This recommender strategy can be considered a proof of concept for operation-based model recommenders, although we are certain that more beneficial content depends on each individual deployment scenario. Because of that, the HERMES Demo IDE we provide comes with only a few `Models`.

All the recommender strategies introduced above take inputs from our query box and work reactively, although model operation sequences can alter them to work proactively. QuickPaste and QuickModel can use selected or created elements as queries to extend existing content. Further, our WordWeb Online recommender strategy can leverage the same and provide results for a selected element. In fact, all these strategies exist, but present their results in a different UI. Only QuickText is not available as a proactive recommender strategy.

5.3. Overall Quality Discussion

Thus far, we have shown that our approach from chapter 3 with a semi-formal notation could serve as the basis for an implementation (cf. chapter 4 (p. 151)), as a language (cf. start of chapter 4 (p. 151) and bullet points in chapter 5 (p. 165)), and as a notation for discussion or development (cf. chapter 4 (p. 151) and Appendix A (p. 195)). Further, we were able to demonstrate that reuse as a process can be subdivided into the components we explained based on operation-based models. This implements not only a single aspect of reuse, but its entirety, which some might denote as the lifecycle of model reuse. Moreover, we found and introduced higher-level solutions in the form of schema for model recommendation production and candidate generation, which, combined with our classification of model persistence, enables precise classification for knowledge-based recommender systems. In a nutshell, this makes our approach a cookbook for knowledge-based model recommendation production.

Further, we could show how to realize our approach from chapter 3 as a software prototype that fulfills the requirements we set initially. This means we have created a piece of software that exceeds proof of concept quality, as we showed by the ISO/IEC 250xx SQuaRE series. Surprisingly, we found that our subjective quality assessment differs from those provided by our SonarQube. For example, some things that we consider to have been covered well were marked down by SonarQube. However, the discrepancy goes further and shows that we did not sugar-coat things, because SonarQube provided a positive assessment of some points that we are not so happy about. Still, there is no reason that this should cause us a headache, because we met our requirements bearing in mind that we have provided a prototype; quality is relative, after all.

Finally, one extra point is worth mentioning: We attribute a big share of the reason why

we achieved a fully functional solution to the quality of our development process, which enabled sufficient quality for our approach and prototype by incremental and iterative development with feedback loops. Hence, we considered small evaluative studies to be sufficient for validating our work, although larger field studies would certainly do no harm. Still, we can claim that we met our derived goals for our project, i.e., the meaning of HERMES, and for our research as presented earlier in this chapter.

5.4. Contribution to Scientific Knowledge Base

Given our approach, realization, assessment, and experiences, we should ask what our contribution to the scientific knowledge base is, or how big (✓, ✓, or ✓) a dent in the universe of knowledge it will make [Mat10]. Ideally, this should go well with the challenges stated in section 1.2 (p. 6), our derived requirements noted in section 2.4 (p. 24), and, eventually, the statements taken from this chapter put in a broader perspective. Keep in mind that we set off to ease model reuse and achieved this by means of operation-based models as a foundation for several processing steps leading to knowledge-based recommender systems.

- ✓ *Storage Challenge:* locating and accessing (by: Φ, Ψ).
- ✓ *Representation Challenge:* warehousing and organizing (by: KL, $\mathsf{I}_{\mathsf{KL}}^{\varepsilon}$).
- ✓ *Harvesting Challenge:* identifying and extracting (by: $\mathsf{ID}|_{D'}$, $\mathsf{s} \subseteq \mathsf{m}$, $\langle\!\langle \mathsf{s}_i$).
- ✓ *Evolution Challenge:* changing and improving (by: A_{staged}, $\{-,\circ,+\}$, QG).
- ✓ *Retrieval Challenge:* querying and retrieving (by: MP).

These challenges, taken from section 1.2 (p. 6) and enriched by our assessment in the form of checkmarks and the symbols representing our solution (cf. list of acronyms and symbols from p. 255), approach our requirements from section 2.4 (p. 24), as we discussed in chapter 4 (p. 151) and Appendix A (p. 195). Hence, we are approaching modeling support with model reuse across its entire lifecycle, from storage all the way to reutilization by means of operation-based models ($\exists$). This encompasses the challenges of storage, representation, harvesting, evolution, and retrieval. The most important outcome is a cookbook derived from deployments that comprises schema for model recommendation production (MP), which are based on solutions regarding the challenges and provides an architecture to implement a tailored solution. In more detail, we propose a quality assured knowledge library (KL) capable of sophisticated queries (Φ) that model recommendation production can rely on. This makes our approach a task-oriented knowledge-based recommender system (cf. figure 3.39 (p. 149) detailed in figure 3.36 (p. 141)) working in cycles of recommendation production operations (ϱ_{ana}, ϱ_{gen}, ϱ_{rnk}, and ϱ_{fil}) and capable of fallbacks or filtering depending on whether there are too many or few recommendation items. In fact, it works for knowledge stored as property graphs with little adjustment.

Our deployment experiences played a major role when we decided to consider not only parts of the model reuse lifecycle, but also its entirety. Because parts that are often considered vital but are excluded, such as harvesting and evolution, contributed to stabilized operation-based models, which otherwise would possibly only function for reutilization, and hence break the platform independence or context analysis. As a result, operation-based models not only function as an approach for the EMOF-reflective API and Ecore, but all modeling languages derived from them. This makes operation-based models potential grounds for many modeling languages, both graphical such as UML with class diagrams and BPMN with process diagrams or textual such as xcore or Xtext. Note that the latter bridges operation-based models to DSLs, which is no surprise bearing in mind that, e.g., Xtext uses Ecore models for code generation.

However, this use of models for DSLs, although not apparent to users, points to the inherent data structure that we are working on: (property) graphs. Therefore, we can say that we are working on the reutilization (item production) of graph structures and their meta-information in the form of graphs. Thus, what else can we represent as graphs and gain benefit from reutilizing them? Turning to DSLs, there does not seem to be much of a difference between them and source code. Equally, what distinguishes textual modeling from programming? Concerning the size of items we take into account, i.e., several elements at once, there would not appear to be much difference except for the level of abstraction. This means that the single element recommendations we introduced could be regarded as elementary code completion and model recommendations could be regarded as source code snippets or templates. Hence, the novelty in our approach lies in the extent, which we denote as the relatedness.

Further, from a software engineering point of view, we consider our higher-level operations in model recommendation production (MP) and their underlying engineering as a contribution. This is because of the template characteristics they expose. Some may denote this a referential architecture [Clo+09; TDM10; Mar+15], but we would rather give a humble assessment and say that we came up with a cookbook for model recommender systems backed by and derived from a few realized recommender strategies.

To that end, we avoided pointing too much to the so-called features of recommender systems, because with our sequences of operations and context analysis, it is easy to transfer basic recommender system knowledge. Consider an index for our introduced god-class clustering and leveraging that index by a respective context analysis or using all classes of a model. This is no challenge.

In addition, our operation recording by means of operation-based models also enabled our proactive quality guidance for knowledge libraries. We consider this to be another contribution to the scientific knowledge base, because our approach joins a quality model for models with quality gates and stages for guided model evolution in knowledge libraries. Certainly, this is coupled to models, because the quality model and respective metrics are bound to models and class diagrams. However, the evolution approach we have presented allows for adoption to other artifacts given our quality gates, or stages.

Research on software often stops at the proof of concept stage, and publishing such software is not always desirable, but we went further with our realization because we

needed this experience to mature our approach and the schema in particular. Hence, we provided our entire prototype for free as an open-source project at an early stage. We picked a rather weak license for our source code, the Eclipse Public License (EPL 1.0), which allows companies to use our results in commercial products even without remaining open source. In addition, we put some effort into achieving a long-term supported prototype by choosing the software we rely on carefully. Further, our build infrastructure is based on an Eclipse Project targeted at long-term support. This should allow our software prototype and implementation to last a long time, which we consider beneficial for our framework architecture.

Another contribution is with regard to our deployment experiences, and we should ask why we have elaborated in such detail on recommender strategies that are not real recommenders (RSSE). They were means of getting modelers used to model reuse and making them familiar with our query box. We did so because we found that modelers are as reluctant as any other users to alter their workflows. In particular, we found that to be true when they had to expend extra effort, which is inevitable for our 80% model reuse. We learned this the hard way, because our first deployment, equipped with our WordWeb Online recommender strategy, and our MoCCa recommender strategy were not well accepted. In fact, that this modeling support existed was rather quickly forgotten. However, after we introduced our recommender strategies QuickText and QuickModel, the usage of our query box improved and, after some time, our MoCCa recommender strategy was "rediscovered". In a nutshell, we introduced a low but beneficial-to-pass barrier so that modelers were willing to adjust their workflow, i.e., work culture.

Finally, a contribution that we consider minor and not directly related to our research is the adjustments to the "Processes and Practices for Quality Scientific Software Projects", as we explained in subsection 5.2.1 (cf. [HLN09]).

Lack of contribution: We are aware of our potential contributions, and see some leads as being beneficial despite the interference of controlled and confounding variables. First, we can ask if model reuse pays off and the counting of editing actions, as we discussed in subsection 5.2.3, provides hard evidence supporting Fitts' Law [AZ03]. This is backed by studies concerning model refactoring, which demonstrate the associated benefit [AMT10; Moh+09; RSA13; SU13]. Additionally, we can argue that model reuse improves quality in modeling in two respects: (a) it provides approved quality, and (b) it provides agreed standards. An example is Vorto, which realizes some of our approach for the "Internet of Things" (IoT) [Ecl15c]. Second, we can ask if model reuse is accepted by users. Other than the cultural discussion in subsection 5.2.4, this probably comes down to tool acceptance and data quality. We approached the former as one of our major requirements, and small user tests confirm that we found a reasonable solution. Regarding the latter, any company working with models and trying to attempt model reuse is in charge of data quality, and we provide the necessary means for a good start. Third, we can evaluate the recommender system algorithms using standard datasets and metrics. For regular recommender systems, precision, recall, and so on are such metrics [CKT10; SG11; Ava+14]. Finally, an agreed on set of recorded interactions is required for comparison, i.e., sequences of model operations in our terms [Rob09].

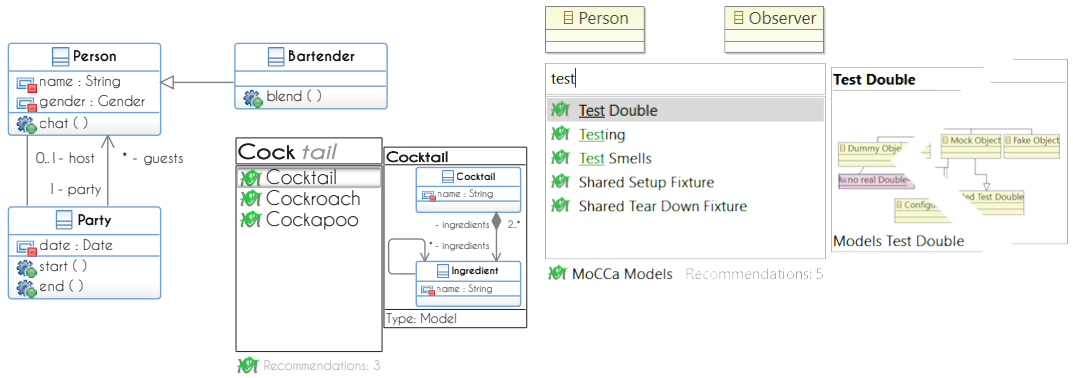
6 The Curtain Falls

Now here I am, a fool for sure,
No wiser than I was before
(Da steh ich nun, ich armer Tor,
und bin so klug als wie zuvor)

J.W.v. Goethe

Contents

The course of this text has researched the surroundings of UML modeling and the reuse of UML models. More precisely, we limited the scope to the more generic EMOF and investigated it by means of class diagram representations. Therefore, we established a vision, defined a set of use cases, and introduced a running example. All of this was founded on existing and derived research questions. In a nutshell, we can claim that we could realize our vision in the software prototype shown in figure 6.1.



(a) Vision (IBM RSA) cf. figure 1.5 (p. 8) (b) Realization (Eclipse) cf. figure 4.6 (p. 157)

Figure 6.1.: Comparing Searchbox and Preview of Model Completion for Class Diagrams

This screenshot proves no more than the existence of a prototype and neglects the concepts that underlie our solution. It combines approaches from data mining, model evolution, and recommender systems, so we were able to develop a coherent solution for model reuse. However, the foundations lie in the two perspectives we defined for models. First, we look at models in a tuple format, with every element comprising either information or relationships. Note that this also provides enough information to derive a graph from the tuple format. Second, we consider models as operation sequences that represent editing actions carried out on the abovementioned tuple.

Each of the domains mentioned above contributes to one part of our approach, and we gave respective summaries to ensure digestible chunks. Now, we provide a conclusive and complementary summary to enable a discussion and outlook. This is why we put subsections 3.1.6, 3.2.6, 3.3.7, 3.4.9 and 3.5.10 (p. 47, 72, 93, 115, 148) together with chapter 4 (p. 151) and chapter 5 (p. 165), to start our discussion from there.

6.1. HERMES Acts Performed

With the idea of model reuse in mind, we started the HERMES project with a vision of model completion (section 1.3 (p. 8)). However, it only demonstrated reactive triggering, and we found out in the course of this research that proactive triggering is vital for software engineering recommender systems [DGL14b]. This was backed up at roughly the same time by Murphy-Hill and Murphy [MM14].

This raises the question of where recommendations should come from and how they should be produced. This leads us to requirements engineering for a model recommender system and the respective research statements, as similarly provided for source code reuse by Janjic, Hummel, and Atkinson [JHA14]. We refined them as follows (cf. section 1.2 (p. 6)):

- *Storage Challenge:* locating and accessing (by: Φ, Ψ).
- *Representation Challenge:* warehousing and organizing (by: KL, $\mathrm{I}_{\mathrm{KL}}^{\epsilon}$).
- *Harvesting Challenge:* identifying and extracting (by: $\mathrm{ID}|_{D'}$, $\mathrm{s} \subseteq \mathrm{m}$, $\langle\!\langle \mathrm{s}_i$).
- *Evolution Challenge:* changing and improving (by: A_{staged}, $\{-,\circ,+\}$, QG).
- *Retrieval Challenge:* querying and retrieving (by: MP).

Certainly, each point could be dealt with in isolation, but our solutions must function seamlessly with one another. This was already indicated by the repeated use of the term model library, which we later refined to knowledge library. Other than that, we needed a sound foundation in terms of a notation for models and respective operations. This came down to the requirement that our notation of models and operations should support and enable the reuse of models. This notation, which we denoted as operation-based models, allowed an approach suitable for dealing with potential information overflow next to the abovementioned research questions. Note that these research questions deliberately omitted one requirement for the software prototype, because it could not interfere with the underlying approach: Usability.

Let us put a pin into usability until later, and first focus on our operation-based models ($\mathfrak{Z} \in \mathcal{M}^{\mathfrak{Z}}$). We introduced them as a notation for sequences of editing operations (σ) on models (m). Therefore, we had a tuple or graph perspective of models to begin with, and altered them successively by sequences of operations. For our later realization, it was useful that each of these operations had a relative in the EMOF reflective API, which has

EMF/Ecore as an implementation. Before that, however, our operation-based models served as a language and sufficiently formal notation on which to found our approach.

Generally, we can say that a model recommendation production works similarly to regular recommender systems, e.g., for online shops. It uses the same processing steps, but applies them to different data. While a recommender system for an online shop analyzes purchased or ranked items to find similar customers and predict the most likely next product or liked item, our model recommendation production uses editing sequences to find possible completions in our knowledge library. Note that the information leveraged for predictions could be denoted simply as an action, because purchasing, ranking, or editing are actions after all. This already tells us that both approaches do not differ as much as we might expect.

We found that the major difference between both recommender systems lay in their nature, which is collaborative filtering for online shops and knowledge-based content for the model knowledge library. We had to use such a knowledge library because collaborative filtering would not have delivered the desired results. The best we could have expected was single additional operations, but we were aiming for model reuse for edited models rather than elementary extensions of them.

Certainly, our knowledge-based content recommender system needed data to work on, and we designed our knowledge library (KL) to be a suitable source of data. It comprised a set of models, each represented as a `Model`, which was organized by a meta-structure forming a graph. This structure provided more than `Model`-to-`Model` relationships, it also allowed categorization and grouping. Hence, models could have neighbors in a meta-structure sense in several respects, and we mirrored this in the granularity and relatedness of model recommendations, as we explain shortly. Further, additional or derived properties of models, i.e., features for regular recommender systems, could be leveraged by the recommendation algorithm realizing a knowledge-based content recommender system.

In more detail, we have broken down a recommendation algorithm into four steps that are similarly used by regular recommender algorithms. First, analysis kicked off the process of model recommendation production and looked into contextual and further information, e.g., recent editing sequences on the model under consideration. Then, model recommendation candidate mrc ($\in \mathcal{MRC}$) generation extracted potential recommendations from our knowledge library; these were ranked in the next step according to their properties. Finally, the ranked model recommendations mr ($\in \mathcal{MR}$) were filtered to give recommendation results r. The notable difference to regular recommender systems lies in how the knowledge library was leveraged as a property graph and how models were treated. Altogether, this meant model recommendation candidates of different granularity and relatedness were produced.

It is important to clarify what is meant by granularity and relatedness. The former refers to potentially beneficial parts of models in our knowledge library, whereas the latter reflects the potential neighbors in our knowledge library. These were indicated by relationships in the abovementioned meta-structure. We distinguished between three kinds of information for direct `Model`-to-`Model` relationships called `Connectors`.

First, we introduced syntactic information for Connectors, which we called cross-links (cl). These contain actual syntactical model information that connects adjacent Models syntactically, e.g., by inheritance or composition relationships, or more. Second, we explained the semantic information for Connectors, which elaborate on the meaning of this relationship. Finally, generic information was provided for the purpose of maintaining the information that the adjacent Models were used together in the past. Other than that, indirect relationships between Models were denoted by Categorys and Groups, which played a role in generating model recommendation candidates with a certain relatedness. As an example, we could assume that a Model was found as a 100% match, i.e., that it was already present in the currently edited model. Hence, this Model from the knowledge library would have been of little use as a model recommendation candidate, so the related Models in our knowledge library became model recommendation candidates regardless of whether they were categorized or grouped with the abovementioned 100% match.

Now, how did this abovementioned 100% Model get into our knowledge library? To make our approach more complete, we looked into approaches that identify potentially beneficial parts and feed them to our knowledge library. We called this harvesting, and provided support for analyzing a model that is currently being edited given a knowledge library. This means we leveraged information from our knowledge library and identified already-known parts ($\mathcal{K}$), so potentially beneficial parts of the currently edited model could be found by means of structural analysis or clustering. As we suspected this would be a highly subjective task, we gave the modeler the last word before new parts were confirmed for the knowledge library. As terms, we talked of submodels (s) as long as the mentioned parts were not stored in the knowledge library, because they could be considered submodels of the currently edited model. After they were persisted in our knowledge library, they became models represented as Models.

Unfortunately, model quality is a rather subjective matter, and models are rarely as stable in the long run as most software is. In fact, as much as models in our knowledge library, software applications undergo change over time, and we investigated this as the issue of model evolution in knowledge libraries. At the same time, this software evolution often leads to declining quality, so we considered model quality in knowledge libraries with proactive quality guidance. This was intended to provide one of three stages for each model, indicating roughly the current quality assessment or the fact that this model had been taken out of service, i.e., been deprecated. Further, it comprised guidance by means of continuous feedback mirroring problems in several respects. Some of them, e.g., syntactic issues, were automatically resolvable, whereas others, e.g., understandability, were a matter for manual investigation through simple reviews.

This concludes our summary from a complementary perspective of a software tool realizing a recommender system that provides an additional reasoning perspective. Certainly, this software prototype exists, as we discussed in chapter 4 (p. 151), and illustrates how our approach came to life to provide strong indicators in an implementable approach. This prototype also served as the basis for validation, feedback, and demonstration, which we used for illustration purposes. In addition, we were able to demonstrate a basic showcase of the demo tool in slightly more than three minutes across three videos. We

took this as an indicator of the simplicity of our tool's usability and integration, despite the elaborate approach. Further, it exposed an architecture that is congruent to our HERMES domain architecture.

At the end of the day, the question arises as to what could possibly remain? The approach we have presented was developed with model reuse in mind, but the bigger picture is quite clear. The essence is about recommender systems for graph-based data, i.e., it presents recommender systems for property graphs. The reason lies in the tiny detail of how our knowledge library models relationships, i.e., the `Connectors`. They are vertices with respect to graph theory, but they are modeling relationships. Certainly, this is no novelty, and emulating property graphs by extra vertices in regular graphs is exactly the same idea. Still, this is uncommon for recommender systems leveraging knowledge libraries modeled as graphs.

6.2. HERMES Acts To Be Performed

The figures that provide an overview of almost each subsection in chapter 3 (figures 3.5, 3.14, 3.23 and 3.33) indicate that many parts are designed for flexibility or extendability. For the model data framework, the versioning is an example, and replacing the versioning software used may open opportunities for further research. A more interesting case is the indexing of the same figure, because we designed the internals to be extendable, as explained in section 3.2 (p. 48). Similarly, this holds true for the model evolution framework, because it is meant to be bound to the model data framework. Hence, the versioning software might be replaced, or, in a more interesting case, metrics and reviews might be altered, as explained in section 3.4 (p. 94). Both of these alterations would require an in-depth understanding, because we do not consider them the most thrilling areas for research. This is different for the frameworks that are responsible for storing or retrieving data. More precisely, figure 3.14 (p. 74) and figure 3.33 (p. 118) already indicate with indexes which parts inherit the potential for further research. First, and most importantly, we see recommender strategies for given contexts and UIs as grounds for experimenting. Second, we see splitter as having similar grounds for experimentation. Certainly, the other parts of both frameworks offer great opportunity and flexibility for new directions, be it new editors, i.e., contexts, or anything else. However, these adjustments or extensions keep the overall approach mostly as is, although they offer some potential for new directions, as we now discuss for each part.

Operation-Based Models: The foundation for our approach lies in operation-based models, and replacing them would be a deal breaker in many respects. Still, deriving a more formal or informal notation could prove beneficial. The former could serve for the mathematical induction that we have generally omitted in the course of this text, while the latter could make operations more readable for human users. Of course, this would mean transforming our notation into a domain-specific language. Compared to other textual notations for models, e.g., EMFText or xcore (textual Ecore) [Hei+09; Ste+08], this would concentrate on changing models.

Storing Models (`store.mdf`): As of now, the model data framework has been deployed for graph databases, relational databases, and version control servers, e.g., git [Tra13]. Although the latter offers only limited functionality, with a distributed system and dedicated servers responsible for versioning, indexing, and meta-information, functionality need not be lacking. Still, as we showed earlier, a realization with a pure directory might be reasonable to get the whole approach deployed (cf. subsection 5.2.4 (p. 179)). Other than that, some parts, which we only mentioned for the sake of completeness, could be explored in more depth. Examples are the `TemplateInformation` [Hu13], the `Example`, or extensions of `LibraryElement`. This is reasonable, because reusable models appear more often than we might think. Consider a DSL modeled in Xtext [EB10]. Underneath, the Xtext framework creates an Ecore model and uses it for code generation with the EMF framework. Therefore, why not investigate these generated models and put them in a knowledge library? Our meta-structure could also be improved, even with our attempt to counter the second system effect [Bro75]. We investigated in this direction, but did not gain the benefit we were hoping for [Fuc11]. Other possible `LibraryElements`, e.g., project documentation, may serve as grounds for successful research [Rei13].

Let us drop the whole knowledge library idea for a moment and consider alternative persistence or data sources. An example for data sources could be common machine learning or regular recommender systems, which could provide fundamentally different data, if fed properly. Still, we found relatively few available data, so we did not go into too much detail about this. Instead, our focus on model reuse required us to produce model recommendations and not individual classes, attributes, methods, and the like. In our terminology, these individuals would roughly map to a small sequence of operations, or one compound operation if we combined one create and add operation in one compound operation.

Harvesting Models (`.harvest.mmf`): As of now, very few marker, splitter, and saver operations have been realized for our model mining framework, and improving on them is a direction worth exploring. This not only means research for UML-related models, but also for other models, which might allow more automation than we achieved. Other than that, more natural language processing, e.g., for camel case notation, better adjusted metrics for splitting, or other approaches for creating submodels are possible research directions. We did not engage in this area because our focus was reutilization in model reuse. Still, without a decently filled knowledge library, any possible testing falls apart and makes the approach much less complete.

What is meant by “other approaches for creating submodels”? We only briefly mentioned that the general problem of finding submodels could be seen as a (bicriteria) optimization or a constraint satisfaction problem, but we omitted the formalization. Still, any relevant solver needs a formalization to be able to work on these problems.

Evolving Models (`evolve.mef`): We bound our model evolution framework tightly to our data framework, and locked some settings that might require further exploration. For example, the quality model we use is not easily adjustable. Certainly, metrics and reviews are designed to be extensible, but editing support for the quality model would be desirable [Hil10]. This becomes more apparent for other models or `LibraryElements`,

e.g., BPMN models. This would even require a different set of metrics. Finally, our investigations on generations were not on a par with their possibilities. The concept proved beneficial for the recommender algorithm, but remains a runtime concept that has not been fully explored. This is also true for quality changes occurring in generations.

Reusing Models (`reuse.mrf`): The current status of our model recommender framework is a job scheduling mechanism for a recommender domain. Thus, it holds together and moderates between several recommender strategies, contexts, and UIs. In particular, the first opens a playground for research, because it allows greater focus on the actual algorithm producing model recommendations. Earlier, we provided and discussed our experiences, but many alterations and extensions are possible. This is not only true for algorithms working on the given environment, i.e., making different uses of model data, meta-information, contextual information, and so forth, but also for alterations to contextual information or the leveraged knowledge library. While the former could use research conducted with interaction data or the Mylyn project, e.g., degree of user interest model, [MFR14; KM05; KM06; MMA14], the latter could explore machine learning or regular recommender systems, as we mentioned above.

One direction that is certainly worth exploring in practice concerns the link between models and source code. Often, source code generated from models is only linked to libraries at a later point, and this is done manually (see generation gap)! However, EMF offers an element denoted `EDataType` that can link to arbitrary classes in libraries. Hence, consider recommended models that not only provide modeled classes, but also provide links to existing libraries that support a later implementation. Examples could be login mechanisms or date calculations, as mentioned on the very first pages of this text. Hence, an element `EDataType` representing `DateTime` from the Joda-Time library (`org.joda.time`)[1] could immediately provide its functionality.

Next to additional data, improved recommender approaches could include so-called community aspects. These community-supported recommender systems leverage ratings or likes, and we implemented likes as a means of social quality for our models in our knowledge library. As of now, we have not used this much in our model recommendations production. Still, it is not difficult to integrate “likes” into the ranking, because the foundation is simple. The general train of thought, however, requires some fundamental changes if applied. This is because of the way recommender systems evaluate such information. For example, these systems could try to assess the likelihood of an item being picked given a certain precondition. This is conditional probability as formulated by Bayes, and is not considered sophisticated mathematics. For lack of data, we did not investigate this further, but we found out that simple approaches were often only marginally outperformed by more complex algorithms, just as naïve Bayes classification performs reasonably well compared with more recent regular probabilistic recommender approaches [Jan+11]. However, for performance measures of recommender systems, the known metrics require full application for an agreed set of data. As of now, there is neither the data nor an agreement on suitable accuracy and error metrics [Ava+14; STC14].

[1] Note that with Java SE 8, it is encouraged to migrate to its `java.time` library (see JSR-310)

Other directions for research would continue GUI designs [DGL14b]. We implemented many GUIs that have not been mentioned in the course of this text, but we did no user experience testing on them. This is certainly worth exploring from a human–computer interaction point of view. Other techniques that we investigated include some so-called quick-fixes, which, at a key-stroke, repair issues. We omitted further investigations because we see these fixes as being too small to be reuse. They are single actions for completing or repairing models. Second, the drag'n'complete functionality, which we have already discussed [DGL14b], is an option for user interaction, and the only reason we did not investigate this further is the technology. We succeeded with a proof of concept for an IBM Rational Software Architect, because the basic functionality is available already, but for our EcoreTools/Sirius-rooted software prototype, this would have meant a lot of back-porting.

Software Prototype: Our software prototype experienced several deployments and appeared sufficient for a prototype. Certainly, there are at least two things worth investigating: First, development in the Eclipse ecosystem is fast and new projects emerge constantly. This has resulted in recent changes to the GUIs with different handling of models. We developed a proof of concept for these new GUIs, but, as of now, the implementation is not fully functional. Second, another glimpse at the software prototype reveals a template structure that recurred for certain kinds of information systems. This concerns our harvesting, data, and reuse framework, which similarly occurs in software architectures for DSS in the early 1980s and later [Spr80]. For our case, we need to put these three together with our concept, language, and extension points (cf. figure 3.14 (p. 74) and figure 3.33 (p. 118)) to gain what some might call a software referential architecture. In particular, the options mentioned for the harvesting and reuse framework, i.e., their interfaces, explain why this might be a software referential architecture, as many believe [Clo+09; TDM10; Mar+15].

Field Studies: We conducted field studies as a means for testing our software, but not for adding another layer to our evaluation stack. Certainly, this harmed the ability to conduct an entire evaluation, and this offers potential for research, not only for UML-related models, but also for BPMN, AUTOSAR, or other types of models. Our short peek at workflows and related approaches for business process modeling showed that these environments also have reuse needs [Mic+15]. Unfortunately, one issue must be addressed first: Although research communities do not get tired celebrating their MDE success stories [Mus+14], our experience does not back up these findings, but rather underlines their conclusion that it is "still a niche technology" [Mus+14]. In fact, we found that MDE is rarely applied, raising the question of why that is. Some answers might be obvious, while others are not [Mus+14]. Furthermore, we need to ask who keeps repositories for collections of models? Only they provide grounds for field studies.

Methodology - HAM: Any concept with associated principles should be surrounded with tools, languages, and methodologies [LL10, System Triangle]. With respect to concepts, principles, tools, and languages, we submitted a working package that some might call holistic, but is lacking in terms of a supporting methodology. This does not mean that there is no methodology, but that, for brevity, we omitted to introduce HAM, our HERMES Agile

Modeling. Additionally, HAM is not extensive enough to be considered as a real scientific contribution. It is about reusing models after all, and many modeling methodologies, e.g., the Unified Modeling Methodology (UMM), discuss this at length. Hence, structure, tasks, roles, work products, and processes would fail to surprise you (note that this is Software & Systems Process Engineering Metamodel (SPEM) 2.0 notation).
Data and Model Repositories: The amount of available data is crucial for information systems intended for reuse, and we have mentioned some publicly available model repositories and libraries (cf. subsection 3.2.5 (p. 68)). None of these is suitable for model reuse, as we already pointed out. Hence, a vacuum exists with respect to publicly available model repositories or better knowledge libraries. However, we are aware that models are often considered vital company assets worth keeping secret, so maybe the modeling culture needs to change, as it did for source code. For now, some models are discoverable through Internet searches.

6.3. Some Final Notes

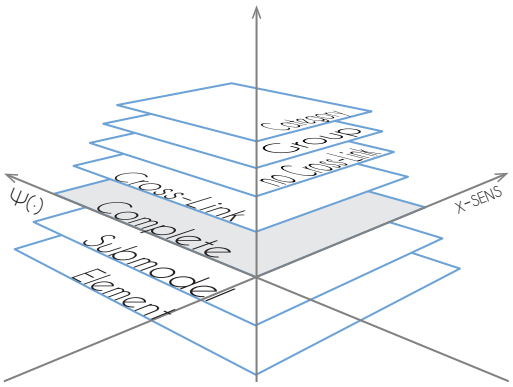
As a final note on modeling, we agree that it has been around for centuries, but that, as an engineering discipline, it is still in its infancy. We think so because it is still considered as an art, but "[f]or conceptual modelling to progress from an 'art' to an engineering discipline, quality standards need to be defined, agreed and applied in practice" [Moo05]. We hope that we have contributed in this respect, because reuse is widely agreed to be a chance for quality improvement. As a final note on models, we found that they are rarely shared with communities and that there is little discussion about this. This closed-source manner is sometimes mentioned as a hindrance to model-driven development, and not only does it put this work ahead of time, but also endangers model-driven development. We are aware that model-driven development comes at a certain cost, but we have also experienced great benefits from it. In some respects, the work preceding this research should have ensured this cultural change as well as publicly available model repositories or knowledge libraries, should there not have been many of them already (see subsection 3.2.5 (p. 68) and [Mus+14]). Thus, the chances are that this needs more "disruptive innovation" [Chr11].

As a final note on recommender systems, we take a step back and look at the greater consequences of using search engines and recommender systems. Fisher, Goddu, and Keil recently found that recommender systems and search engines can result in an overestimation of our own knowledge [FGK15]. This might not be considered harmful at first, but a more detailed look unveils that this induces dependence on such systems—not just in respect of information, but also in its retrieval. Hence, as much as these systems are means for easing everyday life, they manipulate us at the same time, because they draw conclusions for us or sometimes persuade us [YGZ13]. Certainly, this is not a new issue, and artificial intelligence and other domains in computer science have been down that road. However, it always makes sense to indulge in these discussions: after all, we must agree that "I doubt, therefore I think, therefore I am" [Hermes Ref. Space Up ☺].

A Guidelines for Item Ranking

In section 3.5 (p. 117), we established model recommendation production as a sequence of operations, as shown in equation (A.1), and introduced model recommendation item properties that we turned into dimensions of model recommendation production.

$$\text{MP} : \varrho_{fil} \circ \varrho_{rnk} \circ \bigcirc_{i=0}^{n\in\mathbb{N}} \varrho_{gen_i} \circ \varrho_{ana} \quad \text{(equal to equation (3.76) (p. 120))} \tag{A.1}$$



Dimensions of Model Recommendation Production (figure 3.35 (p. 140))

Achieving proper ranking results from ϱ_{rnk} requires insights from candidate generation, i.e., we instantiate a sentence from subsection 3.5.2 (p. 118) while neglecting its impact:

> For a given context (sensitivity), we query our knowledge library (granularity, relatedness), generate model recommendation candidates (extent), and possibly apply them (impact).

Configuration: We can denote such an instantiation as a vector of leveraged data:

$$[\tau, (\Psi(|_{\text{ID}}), ...), (\varrho_{gen}^{X-\text{SENS}}, ...), (\varrho_{gen}^{\text{ID}}, ...), f_{rnk}, \text{c-deg}_{\text{MAX}}]$$

An example for reactive model recommendation production configuration looks like:

$$[\text{reactive}, (\Psi(|_{\text{WModel}}), \Psi(|_{\text{Name}})), (\varrho_{gen}^{\text{BASIC-SENS}}, \varrho_{gen}^{\text{SYNO-SENS}}), (\varrho_{gen}^{\text{C}}, \varrho_{gen}^{\text{CH}^+}, \varrho_{gen}^{\text{CH}^-}), f_{rnk}^{(1)}, 0.8]$$

This uses the querying mechanism of basic means and synonyms as a fallback. Further, model recommendation candidates are only produced in a size given by the knowledge library with direct hits in `Models` and by `Connectors`, no matter whether a cross-link is present. For ranking purposes, a function is given and the completion degree must not exceed 80% of what is already on the modeling canvas. Another example meant for proactive model recommendation production is:

$$[\text{proactive}, (\Psi(|_{\text{WModel}}), \Psi(|_{\text{Name}})), (\varrho_{gen}^{\text{ISO-SENS}}, \varrho_{gen}^{\text{TERMS-SENS}}), (\varrho_{gen}^{\text{C}}, \varrho_{gen}^{\text{CH}^+}, \varrho_{gen}^{\text{CH}^-}), f_{rnk}^{(2)}, 0.8]$$

This queries by means of indexed words from models and names provided by `Models`. Therefore, it relies on a multistep query for isomorphic querying or term sensitivity as a fallback. Results are of a size specified by the knowledge library and include `Models` related by cross-links. Admittedly, these resulting items will be very large for a proactive model recommendation production. Note that, in both cases, the ranking function is not specified, and we discuss this shortly.

Ranking: As difficult as ranking without ratings is [SS11], pseudocode 3.6 (p. 135) and equation (3.115) (p. 133) provide a starting point, but neglect many aspects that can and should be taken into account. Note that this concerns the order for scaling weights, which we introduced in equation (3.121) (p. 135), and the data at hand.

The purpose of ranking stems from the need to order candidates, and we transfer this to a mapping from model recommendation candidates to a real number co-domain between zero and one. To do so, we take into account the strength of indexes, sensitivity, and extent as given in a configuration. Note that we discussed this earlier and depicted these dimensions in appendix A. In some respects, we could talk of strong and weak items expressing the index they result from or explicit and implicit items denoting whether they are direct matches or result from synonyms.

Hence, we require an assessment for all dimensions. The baseline for this is the completion degree. The default concerns the model currently being edited and an extent of “complete” recommendation candidates. In other cases, a sensitivity-related completion degree is the baseline. This is roughly the same as before, but adjusted with a factor. This is because the sensitivity introduced in equations (3.93) to (3.98) on page 128 can be less accurate. For example, a query employing synonyms should be ranked lower than one using basic sensitivity, even if the same number of matching terms are found. This means that an exact completion degree is calculated for the original model with synonyms as above, but the result is reduced by a factor. Hence, our extent property requires an additional factor, because a complete recommendation candidate may be re-asked to produce candidates of the same category. Certainly, we need another factor to adjust for the strength of that extent. Note that the order is descending from complete candidates to either element candidates or category candidates in cases of reactive model recommendation production. In the case of proactive model recommendation production, the descent is ordered from element to complete. Finally, the strength of an employed index is to be taken into account if the default index is abandoned. The reason for this lies in the fact that a used index can leverage manually added meta-information, e.g., from a description.

With these distinctions, we can discuss functions to provide factors between zero and one for adjusting the completion degrees to become sensitivity completion degrees, index strengths, and extent strengths. While exact values are often difficult, a good starting point is a simple quadratic ($x \mapsto 1 - x^2$ cf. figure A.1a) or an exponential function, as depicted in figure A.1. The latter ($(x, y) \mapsto e^{-x^2-y^2}$ cf. figure A.1b) shows that, for optimal values at the origin, a maximum of one is reached in the middle and an equal decrease occurs in every direction, i.e., axis. This is the behavior we are looking for, so we need to

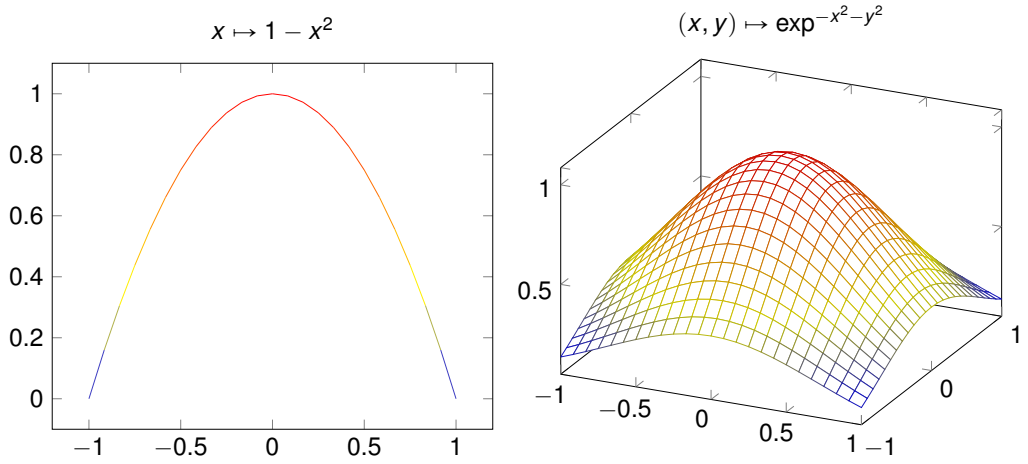
map our inputs accordingly. For example, we can use the order of the given indexes and extents to map each to an interval between zero and one, starting at zero and continuing equidistantly.

Altogether, we obtain a template algorithm as depicted in pseudocode A.1. Note that this scaling has an incompatible interface compared with line 14 in pseudocode 3.6 (p. 135).

```
1  // r is (ε_M^id, ε_M, QUERYTERM) with format (E_M, E_M, str)
2  double ranking(ModelRecommendationCandidate r){
3      // is "Complete" candidate, i.e., not Element, ..., Category
4      if (r.candidate == r.originalModel) {
5          // could be filtered eventually if ⩾ c-deg_MAX
6          return completionDegree(r.candidate);
7      else {// is Element, Submodel, Connector, Group, Category
8          return sensitivityCompletionDegree(r.originalModel) *
9                 indexStrengthOf(r) *
10                extentStrength(r.candidate, r.originalModel);
11     }
12 }
```

Pseudocode A.1: Ranking Candidate

Finally, recommendations should offer a “surprise” to some extent [RW14]. Hence, it could be a good idea to provide one or more random items that will not be shown to the user, because the number of items for a presentation has already been exceeded. This is a dangerous endeavor, because these items must be further explained to the user so that they do not lose confidence in the system [YGZ13].



(a) Quadratic: Two Dimensional

(b) Exponential: Three Dimensional

Figure A.1.: Example for Scaling Functions

MRF Classes

B

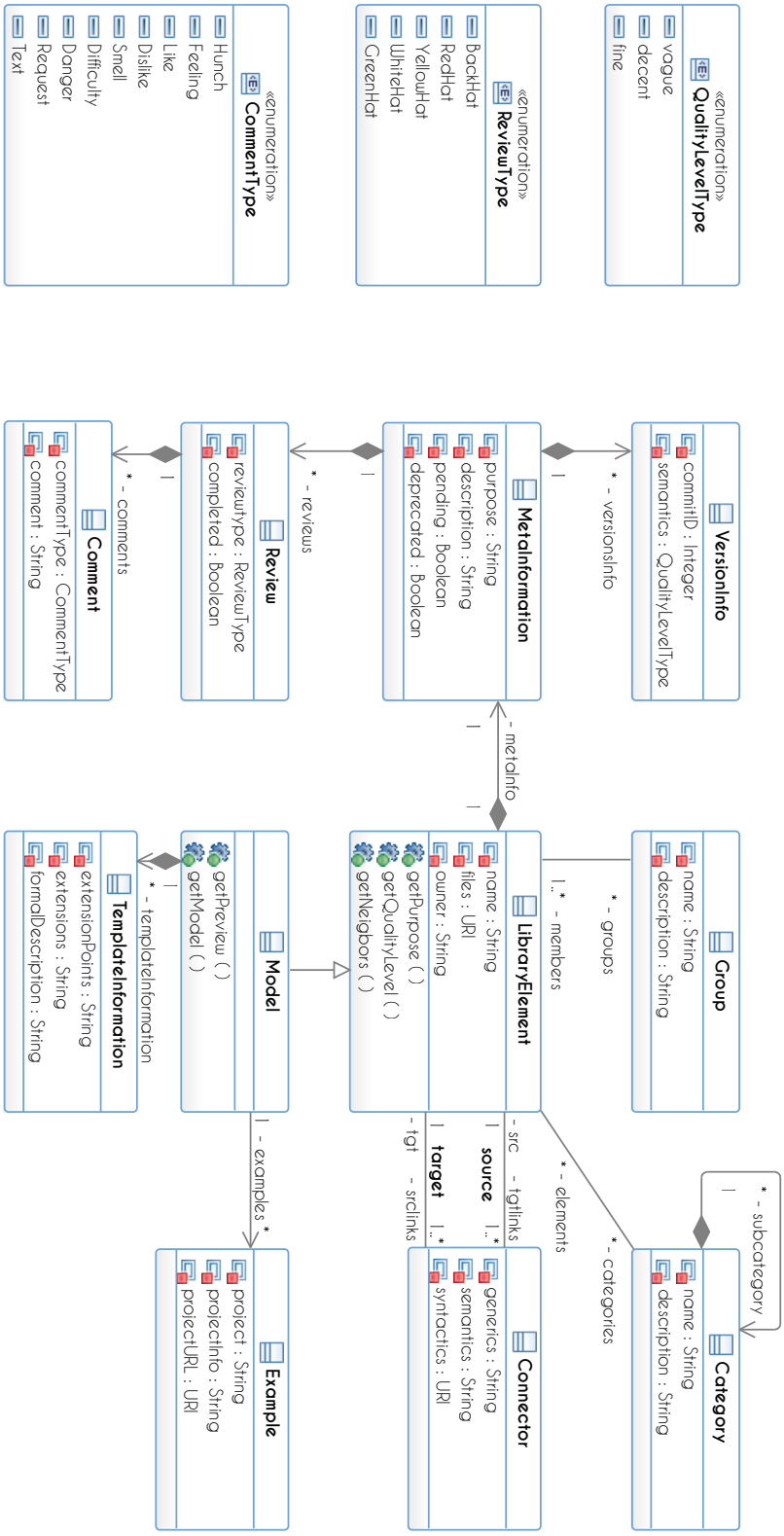


Figure B.1.: MDF: Complete Knowledge Library (similar to [GL13] see pages 51 and 111)

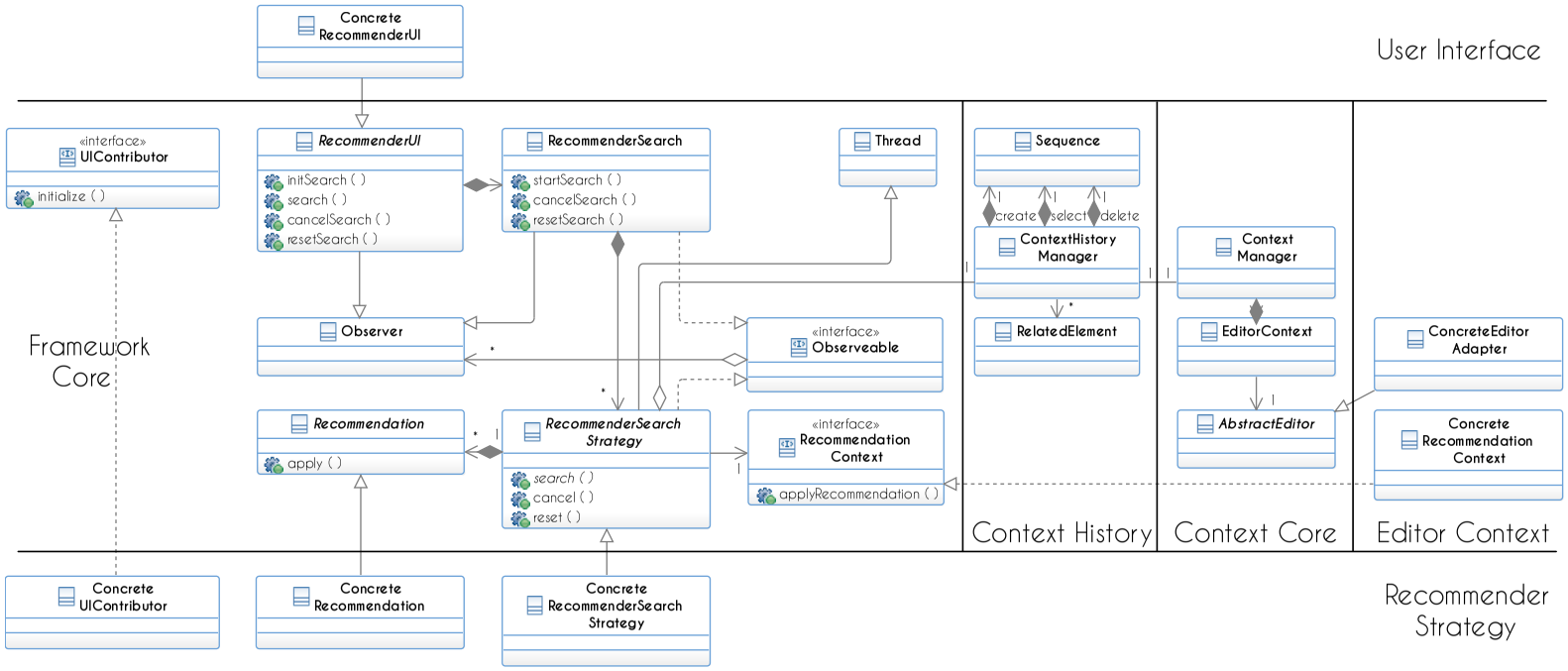


Figure B.2.: MRF: Complete Recommender Framework (extended version of [DGL14a] see pages 163 and 164)

Registered Trademarks

BPMN™ is a registered trademark of Object Management Group, Inc.
Built on Eclipse® is a trademark of Eclipse Foundation, Inc.
Docker™ is a registered trademark of Docker, Inc.
Eclipse® is a trademark of Eclipse Foundation, Inc.
Hibernate™ is a registered trademark of Red Hat, Inc.
Hudson™ is a registered trademark of Oracle, Inc.
IBM® is a trademark of International Business Machines Corporation
Java™ is a registered trademark of Oracle, Inc.
Joda-Time© is a registered trademark of Joda.org
Lucene™ is a registered trademark of the Apache Software Foundation
MATLAB® is a registered trademark of The MathWorks, Inc.
Maven™ is a registered trademark of the Apache Software Foundation
MySQL® is a registered trademark of Oracle, Inc.
MOF™ is a registered trademark of Object Management Group, Inc.
Neo Technology® is a registered trademark of Neo Technology, Inc.
Neo4j® is a registered trademark of Neo Technology, Inc.
OCL™ is a registered trademark of Object Management Group, Inc.
OMG® is a registered trademark of Object Management Group, Inc.
Rational® is a trademark of International Business Machines Corporation
Simulink® is a registered trademark of The MathWorks, Inc.
SonarQube™ is a registered trademark of Sonatype, Inc.
Sonatype Nexus OSS™ is a registered trademark of Sonatype, Inc.
SPEM™ is a registered trademark of Object Management Group, Inc.
TinkerPop Rexster™ is a registered trademark of the Apache Software Foundation
UML® is a registered trademark of Object Management Group, Inc.
VirtualBox™ is a registered trademark of Oracle, Inc.
XMI® is a registered trademark of Object Management Group, Inc.

Bibliography

[Aal11] Aalst, Wil van der. *Process Mining: Discovery, conformance and enhancement of business processes*. Berlin, Heidelberg, and New York: Springer, 2011. ISBN: 978-3-642-19344-6. DOI: 10.1007/978-3-642-19345-3 (cited on page 73).

[AB01] D. H. Akehurst and B. Bordbar. “On Querying UML Data Models with OCL”. In: *UML 2001 – The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. Ed. by G. Goos et al. Vol. 2185. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 91–103. ISBN: 978-3-540-42667-7. DOI: 10.1007/3-540-45441-1_8 (cited on page 70).

[ACS13] A. Ampatzoglou, S. Charalampidou, and I. Stamelos. “Research state of the art on GoF design patterns: A mapping study”. In: *Journal of Systems and Software* 86.7 (2013), pp. 1945–1964. ISSN: 01641212. DOI: 10.1016/j.jss.2013.03.063 (cited on page 48).

[All+06] F. Allilaire et al. “Global Model Management In Eclipse GMT/AM3”. In: *Proceedings of the Eclipse Technology eXchange workshop (eTX) at the ECOOP 2006 Conference*. Ed. by D. Thomas. Vol. 4067. Lecture notes in computer science. 2006 (cited on pages 48, 72, 113).

[Alt+07] K. Altmanninger et al. “Semantically Enhanced Conflict Detection between Model Versions in SMoVer by Example”. In: *Proceedings of the int. Workshop on Semantic-Based Software Development*. 2007 (cited on page 69).

[Alt+08] K. Altmanninger et al. “AMOR - Towards Adaptable Model Versioning”. In: *1st International Workshop on Model Co-Evolution and Consistency Management (MCCM’08), Workshop at MODELS’08*. ACM, 2008, pp. 1–7 (cited on pages 2, 69).

[Alt+09] K. Altmanninger et al. *Why Model Versioning Research is Needed!? An Experience Report*. Denver, USA, 2009 (cited on page 69).

[Alt08] K. Altmanninger. “Models in Conflict – Towards a Semantically Enhanced Version Control System for Models”. In: *Models in Software Engineering*. Ed. by H. Giese. Vol. 5002. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 293–304. ISBN: 978-3-540-69069-6. DOI: 10.1007/978-3-540-69073-3_31 (cited on page 69).

[Ama+11] X. Amatriain et al. “Data Mining Methods for Recommender Systems”. In: *Recommender Systems Handbook*. Ed. by F. Ricci et al. Boston, MA: Springer US, 2011, pp. 39–71. ISBN: 978-0387858197. DOI: 10.1007/978-0-387-85820-3_2 (cited on page 19).

[AMT10] T. Arendt, F. Mantz, and G. Taentzer. “EMF Refactor Specification and Application of Model Refactorings within the Eclipse Modeling Framework”. In: *Proceedings of 9th BElgian-NEtherlands software eVOLution seminar*. 2010 (cited on pages 114, 184).

[AP07] M. Alanen and I. Porres. “A metamodeling language supporting subset and union properties”. In: *Software & Systems Modeling* 7.1 (2007), pp. 103–124. ISSN: 1619-1366. DOI: 10.1007/s10270-007-0049-9 (cited on page 41).

[Arb11] T. Arbuckle. “Studying software evolution using artefacts’ shared information content”. In: *Science of Computer Programming* 76.12 (2011), pp. 1078–1097. ISSN: 01676423. DOI: 10.1016/j.scico.2010.11.005 (cited on pages 101, 103, 144).

[AS06] S. W. Ambler and P. J. Sadalage. *Refactoring Databases: Evolutionary Database Design*. The Addison-Wesley signature series. Upper Saddle River, NJ: Addison Wesley, 2006. ISBN: 0321293533 (cited on pages 44, 45).

[AS08] C. Amelunxen and A. Schürr. “Formalising model transformation rules for UML/MOF 2”. In: *IET Software* 2.3 (2008), p. 204. ISSN: 17518806. DOI: 10.1049/iet-sen:20070076 (cited on pages 32, 43).

[Ast02] D. Astels. “Refactoring With UML”. In: *Proceedings of the 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering*. 2002, pp. 67–70 (cited on page 115).

[AST10] T. Arendt, P. Stepiena, and G. Taentzer. “EMF Metrics Specification and Calculation of Model Metrics within the Eclipse Modeling Framework”. In: *Proceedings of 9th BElgian-NEtherlands software eVOLution seminar*. 2010 (cited on page 114).

[AT11] G. Adomavicius and A. Tuzhilin. “Context-Aware Recommender Systems”. In: *Recommender Systems Handbook*. Ed. by F. Ricci et al. Boston, MA: Springer US, 2011, pp. 217–253. ISBN: 978-0387858197. DOI: 10.1007/978-0-387-85820-3_7 (cited on pages 123, 126, 136, 140).

[AT12] T. Arendt and G. Taentzer. “Integration of Smells and Refactorings within the Eclipse Modeling Framework”. In: *Fifth Workshop on Refactoring Tools (WRT 2012)*. New York, NY: ACM, 2012, pp. 8–15. DOI: 10.1145/2328876.2328878 (cited on page 114).

[AT13] T. Arendt and G. Taentzer. “A tool environment for quality assurance based on the Eclipse Modeling Framework”. In: *Automated Software Engineering* 20.2 (2013), pp. 141–184. ISSN: 0928-8910. DOI: 10.1007/s10515-012-0114-7 (cited on page 114).

[AUv12] R. Accorsi, M. Ullrich, and van der Aalst, Wil M. P. “Process Mining”. In: *Informatik-Spektrum* 35.5 (2012), pp. 354–359. ISSN: 0170-6012. DOI: 10.1007/s00287-012-0641-4 (cited on page 73).

[Ava+14] I. Avazpour et al. “Dimensions and Metrics for Evaluating Recommendation Systems”. In: *Recommendation Systems in Software Engineering*. Ed. by M. P. Robillard et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 245–273. ISBN: 978-3-642-45134-8. DOI: 10.1007/978-3-642-45135-5_10 (cited on pages 184, 191).

[AW12] C. C. Aggarwal and H. Wang, eds. *Managing and Mining Graph Data*. Previously published in hardcover. New York, NY: Springer-Verlag New York Inc, 2012. ISBN: 978-1461425601. DOI: 10.1007/978-1-4419-6045-0 (cited on pages 9, 13, 19, 73, 89, 93).

[AZ03] J. Accot and S. Zhai. “Refining Fitts’ law models for bivariate pointing”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Ed. by G. Cockton. New York, NY: ACM, 2003, pp. 193–200. ISBN: 1-58113-630-7. DOI: 10.1145/642611.642646 (cited on pages 179, 184).

[BA96] S. A. Bohner and R. S. Arnold. *Software Change Impact Analysis*. Los Alamitos, Calif: IEEE Computer Society Press, 1996. ISBN: 9780818673849 (cited on page 108).

[Bal01] H. Balzert. *Lehrbuch der Software-Technik*. Heidelberg: Spektrum Akademischer Verlag, 2001. ISBN: 3827400651 (cited on page 21).

[Ban+87] J. Banerjee et al. “Semantics and Implementation of Schema Evolution in Object-Oriented Databases”. In: *ACM SIGMOD Record* 16.3 (1987), pp. 311–322. ISSN: 01635808. DOI: 10.1145/38714.38748 (cited on page 45).

[Ban98] J. Banks. *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice*. A Wiley-Interscience publication. New York: Wiley and Co-published by Engineering & Management Press, 1998. ISBN: 9780471134039. DOI: 10.1002/9780470172445 (cited on page 160).

[Bas+14] F. Basciani et al. *MDEForge: an extensible Web-based modeling platform*. 2014. URL: http://ceur-ws.org/Vol-1242/paper10.pdf (cited on pages 2, 70).

[Bas92] V. R. Basili. *Software Modeling and Measurement: The Goal/Question/Metric Paradigm*. College Park, MD, USA, 1992 (cited on page 146).

[BB05] M. Bazire and P. Brézillon. “Understanding Context Before Using It”. In: *Modeling and Using Context*. Ed. by D. Hutchison et al. Vol. 3554. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 29–40. ISBN: 978-3-540-26924-3. DOI: 10.1007/11508373_3 (cited on page 124).

[BB91] B. H. Barns and T. B. Bollinger. “Making Reuse Cost-Effective”. In: *IEEE Software* 8.1 (1991), pp. 13–24. ISSN: 0740-7459. DOI: 10.1109/52.62928 (cited on page 4).

[BBF10] A. Bozzon, M. Brambilla, and P. Fraternali. “Searching Repositories of Web Application Models”. In: *Web Engineering*. Ed. by B. Benatallah et al. Vol. 6189. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–15. ISBN: 978-3-642-13910-9. DOI: 10.1007/978-3-642-13911-6_1 (cited on page 70).

[BCG05] D. Berardi, D. Calvanese, and G. d. Giacomo. “Reasoning on UML class diagrams”. In: *Artificial Intelligence* 168.1-2 (2005), pp. 70–118. ISSN: 00043-702. DOI: 10.1016/j.artint.2005.05.003 (cited on page 147).

[BCR06] M. Broy, M. V. Cengarle, and B. Rumpe. *Towards a System Model for UML: The Structural Data Model*. 2006. URL: http://mediatum.ub.tum.de/doc/1094323/file.pdf (cited on page 43).

[BD02] J. Bansiya and C. G. Davis. “A Hierarchical Model for Object-Oriented Design Quality Assessment”. In: *IEEE Transactions on Software Engineering* 28.1 (2002), pp. 4–17. ISSN: 00985589. DOI: 10.1109/32.979986 (cited on page 114).

[Bel99] M. Belaunde. “A Pragmatic Approach for Building a User-Friendly and Flexible UML Model Repository”. In: *UML'99 - The Unified Modeling Language*. Ed. by R. B. France and B. Rumpe. Vol. 1723. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 188–203. ISBN: 978-3-540-66712-4. DOI: 10.1007/3-540-46852-8_14 (cited on pages 23, 69).

[Ben+14] A. Benelallam et al. “Neo4EMF, A Scalable Persistence Layer for EMF Models”. In: *Modelling Foundations and Applications*. Ed. by D. Hutchison et al. Vol. 8569. Lecture notes in computer science. Cham: Springer International Publishing, 2014, pp. 230–241. ISBN: 978-3-319-09194-5. DOI: 10.1007/978-3-319-09195-2_15 (cited on page 67).

[Bet+10] C. Bettini et al. “A survey of context modelling and reasoning techniques”. In: *Pervasive and Mobile Computing* 6.2 (2010), pp. 161–180. ISSN: 15741192. DOI: 10.1016/j.pmcj.2009.06.002 (cited on pages 13, 123, 125).

[Béz+05] J. Bézivin et al. “Modeling in the Large and Modeling in the Small”. In: *Model Driven Architecture*. Ed. by U. Aßmann, M. Aksit, and A. Rensink. Vol. 3599. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 33–46. ISBN: 978-3-540-28240-2. DOI: 10.1007/11538097_3 (cited on pages 23, 48, 72, 168).

[Béz04] J. Bézivin. “In Search of a Basic Principle for Model Driven Engineering”. In: *Novatica Journal, Special Issue* 5.2 (2004), pp. 21–24 (cited on page 18).

[BG10] M. Becker and V. Gruhn. “Automated Model Grouping”. In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. Ed. by C. Pecheur, J. Andrews, and E. Di Nitto. New York, N.Y.: Association for Computing Machinery, 2010, pp. 493–497. ISBN: 978-1-4503-0116-9. DOI: 10.1145/1858996.1859096 (cited on page 92).

[BG93] T. Berlage and A. Genau. “A Framework for Shared Applications with a Replicated Architecture”. In: *Proceedings of the ACM Symposium on User Interface Software and Technology*. Ed. by S. Hudson et al. New York, NY: ACM Pr., 1993, pp. 249–257. ISBN: 0-89791-628-X. DOI: 10.1145/168642.168668 (cited on pages 32, 42, 44).

[Bha+92] A. Bharadwaj et al. “Model management systems: A survey”. In: *Annals of Operations Research* 38.1 (1992), pp. 17–67. ISSN: 0254-5330. DOI: 10.1007/BF02283650 (cited on page 71).

[Bie10] M. Biehl. “Supporting Model Evolution in Model-Driven Development of Automotive Embedded Systems”. Licentiate Thesis. Stockholm, Sweden: KTH, 2010 (cited on page 113).

[Bis+11] B. Bislimovska et al. “Content-based Search of Model Repositories with Graph Matching Techniques”. In: *Proceeding of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*. Ed. by S. Bajracharya, A. Kuhn, and Y. Ye. IEEE Computer Society, 2011, pp. 5–8. DOI: 10.1145/1985429.1985431 (cited on page 70).

[Bis+14] B. Bislimovska et al. “Textual and Content-Based Search in Repositories of Web Application Models”. In: *ACM Transactions on the Web* 8.2 (2014), pp. 1–47. ISSN: 15591131. DOI: 10.1145/2579991 (cited on page 70).

[Bis06] C. M. Bishop. *Pattern Recognition and Machine Learning*. Information science and statistics. New York: Springer, 2006. ISBN: 0-387-31073-8 (cited on page 13).

[BK14] K. Barmpis and D. S. Kolovos. “Evaluation of Contemporary Graph Databases for Efficient Persistence of Large-Scale Models”. In: *The Journal of Object Technology* 13.3 (2014), pp. 1–26. ISSN: 1660-1769. DOI: 10.5381/jot.2014.13.3.a3 (cited on page 18).

[BK95] J. M. Bieman and B.-K. Kang. “Cohesion and Reuse in an Object-Oriented System”. In: *ACM SIGSOFT Software Engineering Notes* 20.SI (1995), pp. 259–262. ISSN: 01635948. DOI: 10.1145/223427.211856 (cited on page 79).

[BL07] J. Bennett and S. Lanning. “The Netflix Prize”. In: *Proceedings of KDD Cup and Workshop 2007*. ACM, 2007, pp. 3–6 (cited on pages 2, 20, 117).

[BL76] L. A. Belady and M. M. Lehman. “A model of large program development”. In: *IBM Systems Journal* 15.3 (1976), pp. 225–252. ISSN: 0018-8670. DOI: 10.1147/sj.153.0225 (cited on page 94).

[Bla+07] X. Blanc et al. “Detecting Model Inconsistency through Operation-Based Model Construction”. In: *ICSE ’08 The 30th International Conference on Software Engineering*. Ed. by W. Schäfer, M. B. Dwyer, and V. Gruhn. IEEE, 2007, pp. 511–520. DOI: 10.1145/1368088.1368158 (cited on pages 42, 46).

[Bla+09] X. Blanc et al. “Incremental Detection of Model Inconsistencies Based on Model Operations”. In: *Advanced Information Systems Engineering*. Ed. by P. van Eck, J. Gordijn, and R. Wieringa. Vol. 5565. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 32–46. ISBN: 978-3-642-02143-5. DOI: 10.1007/978-3-642-02144-2_8 (cited on pages 42, 46).

[Bla93] R. W. Blanning. “Model management systems: An overview”. In: *Decision Support Systems* 9.1 (1993), pp. 9–18. ISSN: 01679236. DOI: 10.1016/0167-9236(93)90019-Y (cited on page 71).

[BLC08] J. H. Bae, K. Lee, and H. S. Chae. “Modularization of the UML Metamodel Using Model Slicing”. In: *Fifth International Conference on Information Technology*. 2008, pp. 1253–1254. ISBN: 0-7695-3099-0. DOI: 10.1109/ITNG.2008.179 (cited on page 91).

[Blo+11] A. Blouin et al. “Modeling Model Slicers”. In: *Model Driven Engineering Languages and Systems*. Ed. by J. Whittle, T. Clark, and T. Kühne. Vol. 6981. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 62–76. ISBN: 978-3-642-24484-1. DOI: 10.1007/978-3-642-24485-8_6 (cited on page 92).

[Blo+15] A. Blouin et al. “Kompren: modeling and generating model slicers”. In: *Software & Systems Modeling* 14.1 (2015), pp. 321–337. ISSN: 1619-1366. DOI: 10.1007/s10270-012-0300-x (cited on pages 90, 92).

[BM08a] A. Boronat and J. Meseguer. “An Algebraic Semantics for MOF”. In: *Fundamental Approaches to Software Engineering*. Ed. by J. L. Fiadeiro and P. Inverardi. Vol. 4961. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 377–391. ISBN: 978-3-540-78742-6. DOI: 10.1007/978-3-540-78743-3_28 (cited on pages 41, 43).

[BM08b] M. Bruch and M. Mezini. “Improving Code Recommender Systems using Boolean Factor Analysis and Graphical Models”. In: *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering*. Ed. by M. Robillard, R. Walker, and T. Zimmermann. RSSE ’08. New York, NY, USA: ACM, 2008, p. 1. DOI: 10.1145/1454247.1454267 (cited on page 144).

[BMM09] M. Bruch, M. Monperrus, and M. Mezini. “Learning from Examples to Improve Code Completion Systems”. In: *7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. Ed. by H. van Vliet and V. Issarny. 2009, pp. 213–222. DOI: 10.1145/1595696.1595728 (cited on page 48).

[Boh11] F. Bohuschke. “Entwicklung eines interaktiven Verwaltungswerkzeuges für Ecore Bibliotheken”. Bachelor Thesis. Aachen, Germany: RWTH Aachen University, 2011 (cited on page 167).

[Bon99] E. d. Bono. *Six Thinking Hats: Run Better Meetings, Make Faster Decisions*. Revised and updated. London: Penguin, 1999. ISBN: 0141033053 (cited on pages 104, 110, 114).

[Bor+09] M. Born et al. “Auto-completion for Executable Business Process Models”. In: *Business Process Management Workshops*. Ed. by D. Ardagna, M. Mecella, and J. Yang. Vol. 17. Lecture Notes in Business Information Processing. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 510–515. ISBN: 978-3-642-00327-1. DOI: 10.1007/978-3-642-00328-8_51 (cited on page 148).

[Bor07] A. Boronat. “MOMENT: A Formal Framework for MOdel managemMENT”. PhD Thesis. València, Spain: Universitat Politècnica de València, 2007. URL: https://riunet.upv.es/bitstream/handle/10251/1964/tesisUPV2716.pdf (cited on pages 38, 41, 43).

[BR00] K. H. Bennett and V. T. Rajlich. “Software Maintenance and Evolution: a Roadmap”. In: *Proceedings of the Conference on The Future of Software Engineering*. Ed. by A. Finkelstein. New York, NY: ACM, 2000, pp. 73–87. ISBN: 1-58113-253-0. DOI: 10.1145/336512.336534 (cited on pages 94, 113).

[BR05] A. Bobkowska and K. Reszke. “Usability of UML Modeling Tools”. In: *Software Engineering: Evolution and Emerging Technologies*. Ed. by K. Zieliński and T. Szmuc. Vol. v. 130. Frontiers in artificial intelligence and applications. Amsterdam and Washington, DC: IOS Press, 2005, pp. 75–86. ISBN: 1607501422 (cited on page 6).

[BR11] R. Burke and M. Ramezani. “Matching Recommendation Technologies and Domains”. In: *Recommender Systems Handbook*. Ed. by F. Ricci et al. Boston, MA: Springer US, 2011, pp. 367–386. ISBN: 978-0387858197. DOI: 10.1007/978-0-387-85820-3_11 (cited on pages 127, 144, 174).

[BR14] M. Borg and P. Runeson. “Changes, Evolution, and Bugs”. In: *Recommendation Systems in Software Engineering*. Ed. by M. P. Robillard et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 477–509. ISBN: 978-3-642-45134-8. DOI: 10.1007/978-3-642-45135-5_18 (cited on pages 20, 117).

[Bra+08] U. Brandes et al. “On Modularity Clustering”. In: *IEEE Transactions on Knowledge and Data Engineering* 20.2 (2008), pp. 172–188. ISSN: 1041-4347. DOI: 10.1109/TKDE.2007.190689 (cited on page 76).

[Bre74] B. Brecht. “Schriften 5 zur Politik und Gesellschaft 1919 - 1956”. In: *Bertolt Brecht, Gesammelte Werke, Notizen zur Philosophie*. Ed. by E. Hauptmann. Vol. 5. Suhrkamp Verlag, 1974, pp. 168–170 (cited on page 17).

[Brè96] P. Brèche. “Advanced Primitives for Changing Schemas of Object Databases”. In: *Advanced Information Systems Engineering*. Ed. by P. Constanto, J. Mylopoulos, and Y. Vassiliou. Vol. 1080. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 476–495. ISBN: 978-3-540-61292-6. DOI: 10.1007/3-540-61292-0_26 (cited on page 45).

[Bri+06] L. C. Briand et al. “Automated impact analysis of UML models”. In: *Journal of Systems and Software* 79.3 (2006), pp. 339–352. ISSN: 01641212. DOI: 10.1016/j.jss.2005.05.001 (cited on pages 98, 115).

[Bro+09] P. Brosch et al. “An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example”. In: *Model Driven Engineering Languages and Systems*. Ed. by A. Schürr and B. Selic. Vol. 5795. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 271–285. ISBN: 978-3-642-04424-3. DOI: 10.1007/978-3-642-04425-0_20 (cited on page 115).

[Bro75] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Reading, Mass.: Addison-Wesley Pub. Co., 1975. ISBN: 0-201-00650-2. URL: https://archive.org/details/mythicalmanmonth00fred (cited on pages 65, 168, 190).

[Bru08] M. Bruch. “Towards Control-flow Aware Code Recommender Systems”. In: *3rd International Doctoral Symposium on Empirical Software Engineering 2008*. Kaiserslautern, Germany: TU Darmstadt, 2008, pp. 1–6. URL: http://tubiblio.ulb.tu-darmstadt.de/33980/ (cited on page 144).

[Bru12] M. Bruch. “IDE 2.0: Leveraging the Wisdom of the Software Engineering Crowds”. PhD Thesis. Darmstadt: TU Darmstadt, 2012. URL: http://tubiblio.ulb.tu-darmstadt.de/59505/ (cited on pages 2, 9, 20, 22, 144).

[BSF03] M. Boger, T. Sturm, and P. Fragemann. “Refactoring Browser for UML”. In: *Objects, Components, Architectures, Services, and Applications for a Networked World*. Vol. 2591. Lecture notes in computer science. 2003, pp. 366–377. DOI: 10.1007/3-540-36557-5_26 (cited on page 115).

[BSM08] M. Bruch, T. Schäfer, and M. Mezini. “On Evaluating Recommender Systems for API Usages”. In: *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering*. Ed. by M. Robillard, R. Walker, and T. Zimmermann. RSSE ’08. New York, NY, USA: ACM, 2008, pp. 16–20. DOI: 10.1145/1454247.1454254 (cited on page 144).

[Bus+14] J. Busse et al. “Was bedeutet eigentlich Ontologie?” In: *Informatik-Spektrum* 37.4 (2014), pp. 286–297. ISSN: 0170-6012. DOI: 10.1007/s00287-012-0619-2 (cited on page 68).

[Cas95] E. Casais. “Managing Class Evolution in Object-Oriented Systems”. In: *Object-oriented software composition*. Ed. by O. M. Nierstrasz and D. C. Tsichritzis. The Object-oriented series. London: Prentice Hall, 1995, pp. 201–244. ISBN: 0-13-220674-9 (cited on page 45).

[Cha+01] N. Chapin et al. “Types of software evolution and software maintenance”. In: *Journal of Software Maintenance and Evolution: Research and Practice* 13.1 (2001), pp. 3–30. ISSN: 1532-060X. DOI: 10.1002/smr.220 (cited on page 94).

[CHM10] J. Clarkson, R. Hammond, and J. May. *Top Gear: The Cheap Show, Season 14, Episode 7 (2010-01-03)*. 2010. URL: http://www.topgear.com/ (visited on 05/01/2015) (cited on page 257).

[CHN12] M. Chaudron, W. Heijstek, and A. Nugroho. “How effective is UML modeling?” In: *Software & Systems Modeling* 11.4 (2012), pp. 571–580. ISSN: 1619-1366. DOI: 10.1007/s10270-012-0278-4 (cited on page 48).

[Chr11] C. M. Christensen. *The Innovator’s Dilemma: The revolutionary book that will change the way you do business*. 1. Harper Business paperback. New York, NY [u.a.]: Harper Business, 2011. ISBN: 9780062060242 (cited on page 193).

[Cic+08] A. Cicchetti et al. “Automating Co-evolution in Model-Driven Engineering”. In: *Enterprise Distributed Object Computing Conference (EDOC)*. Munich: IEEE, 2008, pp. 222–231. ISBN: 978-0-7695-3373-5. DOI: 10.1109/EDOC.2008.44 (cited on page 94).

[CK94] S. R. Chidamber and C. F. Kemerer. “A Metrics Suite for Object Oriented Design”. In: *IEEE Transactions on Software Engineering* 20.6 (1994), pp. 476–493. ISSN: 00985589. DOI: 10.1109/32.295895 (cited on pages 60, 79).

[CKT10] P. Cremonesi, Y. Koren, and R. Turrin. “Performance of Recommender Algorithms on Top-N Recommendation Tasks”. In: *Proceedings of the fourth ACM conference on Recommender systems*. Ed. by X. Amatriain. New York, NY: ACM, 2010, p. 39. ISBN: 978-1-60558-906-0. DOI: 10.1145/1864708.1864721 (cited on page 184).

[Clo+09] R. Cloutier et al. “The Concept of Reference Architectures”. In: *Systems Engineering* (2009), n/a. ISSN: 10981241. DOI: 10.1002/sys.20129 (cited on pages 183, 192).

[CMZ08] C. A. Curino, H. J. Moon, and C. Zaniolo. “Graceful Database Schema Evolution: the PRISM Workbench”. In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 761–772. ISSN: 21508097. DOI: 10.14778/1453856.1453939 (cited on page 45).

[CO09] M. Cargill and O’Connor, Patrick D. T. *Writing Scientific Research Articles: Strategies and Steps*. Hoboken, N.J: Wiley-Blackwell, 2009. ISBN: 978-1-4051-8619-3 (cited on page 10).

[Cor+01] L. P. Cordella et al. “An Improved Algorithm for Matching Large Graphs”. In: *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*. 2001, pp. 149–159 (cited on page 64).

[Cor+04] L. P. Cordella et al. “A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs”. In: *IEEE Trans. Pattern Anal. Mach. Intell.* 26.10 (2004), pp. 1367–1372. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2004.75 (cited on page 64).

[CS03] U. o. Chicago and P. Staff, eds. *The Chicago Manual of Style: The Essential Guide for Writers, Editors, and Publishers*. 15th Edition. Chicago: University of Chicago Press, 2003. ISBN: 0226104036 (cited on page 10).

[CS09] S. Chacon and B. Straub. *Pro Git*. The expert’s voice in software development. Berkeley, CA and New York: Apress and Distributed to the Book trade worldwide by Springer-Verlag, 2009. ISBN: 978-1430218333 (cited on pages 30, 153).

[Cur+09] C. A. Curino et al. “The PRISM Workwench: Database Schema Evolution without Tears”. In: *25th International Conference on Data Engineering (ICDE)*. IEEE, 2009, pp. 1523–1526. DOI: 10.1109/ICDE.2009.46 (cited on page 45).

[Cyb+98] J. L. Cybulski et al. “Reuse of early life-cycle artifacts: workproducts, methods and tools”. In: *Annals of Software Engineering* 5 (1998), pp. 227–251. ISSN: 10227091. DOI: 10.1023/A:1018983220136 (cited on pages 3, 7, 8, 22).

[Dag+10] B. Dagenais et al. “Moving into a New Software Project Landscape”. In: *ACM/IEEE 32nd International Conference on Software Engineering, 2010*. Ed. by J. Kramer. Piscataway, NJ: IEEE, 2010, p. 275. ISBN: 1605587192. DOI: 10.1145/1806799.1806842 (cited on page 117).

[DGD05] D. Djuric, D. Gaševi, and V. Devedžic. “Ontology Modeling and MDA”. In: *The Journal of Object Technology* 4.1 (2005), p. 109. ISSN: 1660-1769. DOI: 10.5381/jot.2005.4.1.a3 (cited on page 67).

[DGL13] A. Dyck, A. Ganser, and H. Lichter. “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*. CEUR, 2013, pp. 12–21 (cited on pages 9, 10, 101, 118, 153, 157, 159).

[DGL14a] A. Dyck, A. Ganser, and H. Lichter. “A Framework for Model Recommenders – Requirements, Architecture and Tool Support”. In: *Modelsward 2014, Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development, Lisbon, Portugal, 7.-9. January 2014*. SCITEPRESS, 2014, pp. 282–290. ISBN: 978-989-758-007-9. DOI: 10.5220/0004701702820290 (cited on pages 9, 10, 117, 118, 142, 147, 153, 157–160, 163, 169, 201).

[DGL14b] A. Dyck, A. Ganser, and H. Lichter. “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, 2014, pp. 291–299. ISBN: 978-989-758-007-9. DOI: 10.5220/0004701802910299 (cited on pages 6–8, 10, 118, 120, 122, 123, 153, 157, 166, 169, 175, 186, 192).

[DK75] F. DeRemer and H. Kron. “Programming-in-the large versus Programming-in-the-small”. In: *ACM SIGPLAN Notices* 10.6 (1975), pp. 114–121. ISSN: 03621340. DOI: 10.1145/390016.808431 (cited on page 72).

[DMW05] C. R. Dantas, L. G. Murta, and C. Werner. “Consistent Evolution of UML Models by Automatic Detection of Change Traces”. In: *Proceedings of the 8th International Workshop on Principles of Software Evolution*. IEEE, 2005, pp. 144–147. DOI: 10.1109/IWPSE.2005.10 (cited on page 115).

[Dol14] S. Dollase. “Context-History Support for Ecore Library Recommender”. Bachelor Thesis. Aachen, Germany: RWTH Aachen University, 2014 (cited on pages 158, 164, 175).

[DTC13] Davi Viana, Tayana Conte, and Cleidson R. B. de Souza. “Knowledge Transfer between Senior and Novice Software Engineers: A Qualitative Analysis”. In: *The 26th International Conference on Software Engineering and Knowledge Engineering*. Ed. by Marek Reformat. Knowledge Systems Institute Graduate School, 2013, pp. 235–240 (cited on page 8).

[Dyc12a] A. Dyck. *Eclipse Modeling Forums: An “intelligent” autocomplete function for Ecore diagram editors*. 2012. URL: http://www.eclipse.org/forums/index.php/t/354604/ (visited on 09/09/2012) (cited on page 166).

[Dyc12b] A. Dyck. “Recommender System Architecture for Ecore Libraries”. Master Thesis. Aachen, Germany: RWTH Aachen University, 2012 (cited on pages 158, 166).

[EB10] M. Eysholdt and H. Behrens. “Xtext - Implement your Language Faster than the Quick and Dirty way”. In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. Ed. by W. R. Cook. New York, NY: ACM, 2010, p. 307. ISBN: 978-1-4503-0240-1. DOI: 10.1145/1869542.1869625 (cited on pages 50, 190).

[Ecl10] Eclipse Foundation. *CDO (Connected Data Objects) Model Repository*. 2010. URL: https://wiki.eclipse.org/CDO (visited on 05/01/2010) (cited on page 69).

[Ecl12a] Eclipse Foundation. *EMF Index*. 2012. URL: http://www.eclipse.org/proposals/emf-index/ (cited on page 69).

[Ecl12b] Eclipse Foundation. *EMF Search*. 2012. URL: http://wiki.eclipse.org/EMF_Search (cited on page 69).

[Ecl14a] Eclipse Foundation. *Code Recommenders*. 2014. URL: http://www.eclipse.org/recommenders/ (visited on 04/01/2014) (cited on pages 2, 3, 9, 14, 117, 144).

[Ecl14b] Eclipse Foundation. *MoDisco*. 2014. URL: http://www.eclipse.org/MoDisco/ (visited on 04/01/2014) (cited on page 113).

[Ecl15a] Eclipse Foundation. *EMF Query*. 2015. URL: https://projects.eclipse.org/projects/modeling.emf.query (cited on page 69).

[Ecl15b] Eclipse Foundation. *SnipMatch*. 2015. URL: http://www.snipmatch.com/ (visited on 03/02/2015) (cited on pages 144, 145).

[Ecl15c] Eclipse Foundation. *Vorto*. 2015. URL: http://www.eclipse.org/vorto/ (visited on 04/01/2015) (cited on page 184).

[Eic+01] S. G. Eick et al. "Does Code Decay? Assessing the Evidence from Change Management Data". In: *IEEE Transactions on Software Engineering* 27.1 (2001), pp. 1–12. ISSN: 00985589. DOI: 10.1109/32.895984 (cited on page 98).

[Eli+10] S. Elinson et al. *Eclipse Model Repository*. 2010. URL: http://modelrepository.sourceforge.net/ (visited on 04/01/2014) (cited on pages 66, 69, 89).

[Enc09] T. v. Enckevort. "Refactoring UML Models: Using OpenArchitectureWare to measure UML model quality and perform pattern matching on UML models with OCL queries". In: *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. New York, NY: ACM, 2009. ISBN: 978-1-60558-768-4 (cited on page 115).

[FAM06] Fabiana Lorenzi, Ana L. C. Bazzan, and Mara Abel. "An Architecture for a Multiagent Recommender System in Travel Recommendation Scenarios". In: *Proceedings of 3rd Workshop on Model-Based Systems (ECAI 2006)*. Ed. by A. Felfernig and M. Zanker. ECAI, 2006, pp. 88–91. URL: www.inf.ufrgs.br/maslab/pergamus/pubs/lorenzietalecai2006.zip (cited on page 148).

[FBC06] R. B. France, J. Bieman, and B. H. C. Cheng. "Repository for Model Driven Development (ReMoDD)". In: *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems*. Ed. by T. Kühne. Vol. 4364. Lecture notes in computer science. ACM, 2006, pp. 311–317. ISBN: 978-3-540-69488-5. DOI: 10.1007/978-3-540-69489-2_38 (cited on pages 2, 70, 158).

[FCC06] M. Fernandez, I. Cantador, and P. Castells. "CORE: A Tool for Collaborative Ontology Reuse and Evaluation". In: *4th International Workshop on Evaluation of Ontologies for the Web (EON 2006)*. 2006. URL: http://oro.open.ac.uk/28592/ (cited on page 71).

[Fel+13] A. Felfernig et al. “An Overview of Recommender Systems in Requirements Engineering”. In: *Managing Requirements Knowledge*. Ed. by W. Maalej and A. K. Thurimella. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 315–332. ISBN: 978-3-642-34418-3. DOI: 10.1007/978-3-642-34419-0_14 (cited on page 117).

[Fer+95] F. Ferrandina et al. “Schema and Database Evolution in the O2 Object Database System”. In: *VLDB ’95*. Ed. by U. Dayal, Gray, Peter M. D, and S. Nishio. San Francisco, CA: Morgan Kaufmann Publishers, 1995, pp. 170–181. ISBN: 1-55860-379-4 (cited on page 45).

[FGK15] M. Fisher, M. K. Goddu, and F. C. Keil. “Searching for Explanations: How the Internet Inflates Estimates of Internal Knowledge”. In: *Journal of experimental psychology. General* 144.3 (2015), pp. 674–687. ISSN: 1939-2222. DOI: 10.1037/xge0000070 (cited on page 193).

[FGL12] S. R. Foster, W. G. Griswold, and S. Lerner. “WitchDoctor: IDE Support for Real-Time Auto-Completion of Refactorings”. In: *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. IEEE Computer Society Press, 2012, pp. 222–232. DOI: 10.1109/ICSE.2012.6227191 (cited on page 145).

[FI94] W. B. Frakes and S. Isoda. “Success Factors of Systematic Reuse”. In: *IEEE Software* 11.5 (1994), pp. 14–19. ISSN: 0740-7459. DOI: 10.1109/52.311045 (cited on page 4).

[FK05] W. B. Frakes and Kyo Kang. “Software Reuse Research: Status and Future”. In: *IEEE Transactions on Software Engineering* 31.7 (2005), pp. 529–536. ISSN: 00985589. DOI: 10.1109/TSE.2005.85 (cited on pages 2, 22).

[Flo62] R. W. Floyd. “Algorithm 97: Shortest path”. In: *Communications of the ACM* 5.6 (1962), p. 345. ISSN: 00010782. DOI: 10.1145/367766.368168 (cited on pages 79, 88).

[For10] S. Fortunato. “Community detection in graphs”. In: *Physics Reports* 486.3-5 (2010), pp. 75–174. ISSN: 03701573. DOI: 10.1016/j.physrep.2009.11.002 (cited on page 76).

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. The Addison-Wesley object technology series. Reading, MA: Addison-Wesley, 1999. ISBN: 0201485672 (cited on pages 32, 48, 104, 155).

[FP79] R. S. Feldman and T. Prohaska. “The student as Pygmalion: Effect of student expectation on the teacher”. In: *Journal of Educational Psychology* 71.4 (1979), pp. 485–493. ISSN: 0022-0663. DOI: 10.1037/0022-0663.71.4.485 (cited on page 165).

[Fra01] A. U. Frank. “Tiers of ontology and Consistency Constraints in Geographical Information Systems”. In: *International Journal of Geographical Information Science* 15.7 (2001), pp. 667–678. ISSN: 1365-8816. DOI: 10.1080/13658810110061144 (cited on page 125).

[Fra95] F. K. Frantz. “A Taxonomy of Model Abstraction Techniques”. In: *Winter Simulation Conference*. 1995, pp. 1413–1420. ISBN: 0-78033018-8. DOI: 10.1109/WSC.1995.479055 (cited on page 90).

[Fuc11] C. Fuchs. “Developing a Generic Pattern-Based Ecore Library”. Master Thesis. Aachen, Germany: RWTH Aachen University, 2011 (cited on pages 65, 168, 190).

[Fur14] F. J. Furrer. “Eine kurze Geschichte der Ontologie”. In: *Informatik-Spektrum* 37.4 (2014), pp. 308–317. ISSN: 0170-6012. DOI: 10.1007/s00287-012-0642-3 (cited on page 68).

[Gam+95] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley professional computing series. Reading, Mass.: Addison-Wesley, 1995. ISBN: 0-201-63361-2 (cited on pages 21, 28, 38, 48, 52–54, 57, 61, 85, 99, 125, 146, 181).

[Gan+13] A. Ganser et al. “Proactive Quality Guidance for Model Evolution in Model Libraries”. In: *Models and Evolution, Proceedings of the Workshop on Models and Evolution, co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2013)*. Vol. 1090. CEUR Workshop Proceedings. CEUR, 2013, pp. 50–59 (cited on pages 7, 9, 10, 95, 153, 156).

[Gan+16] A. Ganser et al. “Staged model evolution and proactive quality guidance for model libraries”. In: *Software Quality Journal* 24.3 (2016), pp. 675–708. ISSN: 0963-9314. DOI: 10.1007/s11219-015-9298-y (cited on pages 10, 95, 97, 98, 100, 102, 103, 105, 107, 153).

[Gan13a] A. Ganser. *Reusing Domain Engineered Artifacts for Code Generation – The HERMES Project (Harvesting, Evolving, and Reusing Models Easily and Seamlessly)*. 2013. URL: http://goo.gl/4LRdN (cited on page 12).

[Gan13b] A. Ganser. *YouTube: Model Autocompletion Demo*. 2013. URL: http://goo.gl/fqwxl (cited on page 8).

[Gan14a] A. Ganser. *HERMES Demo Session 01 Download and Install*. 2014. URL: http://goo.gl/7ZvU0Y (visited on 10/12/2014) (cited on page 161).

[Gan14b] A. Ganser. *HERMES Demo Session 02 MMF Demo*. 2014. URL: http://goo.gl/VT9N7S (visited on 10/12/2014) (cited on pages 161, 166).

[Gan14c] A. Ganser. *HERMES Demo Session 03 MEF Demo*. 2014. URL: http://goo.gl/vkviQK (visited on 10/12/2014) (cited on pages 161, 166).

[Gan14d] A. Ganser. *HERMES Demo Session 04 MRF Demo*. 2014. URL: http://goo.gl/orDZf7 (visited on 10/12/2014) (cited on pages 161, 166).

[Gan14e] A. Ganser. *HERMES Demo Session 05 MDF Demo*. 2014. URL: https://goo.gl/vbNvk3 (visited on 10/12/2014) (cited on page 161).

[Gan14f] A. Ganser. *The HERMES Project - Demo Videos*. 2014. URL: http://hermes.modelrecommenders.org/demos.html (visited on 10/12/2014) (cited on page 161).

[Gan14g] A. Ganser. *The HERMES Project - Downloads*. Ed. by A. Ganser. 2014. URL: http://hermes.modelrecommenders.org/downloads.html (visited on 10/12/2014) (cited on pages 12, 161, 177, 178).

[Gan14h] A. Ganser. *The HERMES Project: Harvest, Evolve, and Reuse Models Easily and Seamlessly*. Ed. by A. Ganser. 2014. URL: http://hermes.modelrecommenders.org/ (visited on 10/12/2014) (cited on pages 12, 161).

[GAO95] D. Garlan, R. Allen, and J. Ockerbloom. "Architectural Mismatch: Why Reuse Is So Hard". In: *IEEE Software* 12.6 (1995), pp. 17–26. ISSN: 0740-7459. DOI: 10.1109/52.469757 (cited on page 4).

[Gen+03] M. Genero et al. "Building UML Class Diagram Maintainability Prediction Models Based on Early Metrics". In: *9th IEEE International Software Metrics Symposium*. IEEE, 2003, pp. 263–275. ISBN: 0-7695-1987-3. DOI: 10.1109/METRIC.2003.1232473 (cited on page 114).

[GG08] M. W. Godfrey and D. M. German. "The Past, Present, and Future of Software Evolution". In: *Frontiers of Software Maintenance (FoSM)*. IEEE, 2008, pp. 129–138. DOI: 10.1109/FOSM.2008.4659256 (cited on pages 94, 113).

[GL13] A. Ganser and H. Lichter. "Engineering Model Recommender Foundations". In: *Modelsward 2013, Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development, Barcelona, Spain,19.-21- February 2013*. SCITEPRESS, 2013, pp. 135–142. ISBN: 978-989-8565-42-6. DOI: 10.5220/0004320801350142 (cited on pages 7–10, 49, 51, 66, 111, 147, 153, 167, 200).

[Gog+96] J. Goguen et al. "Software Component Search". In: *Journal of Systems Integration* 6.1-2 (1996), pp. 93–134. ISSN: 0925-4676. DOI: 10.1007/BF02262753 (cited on page 68).

[Gom+04] P. Gomes et al. "Using WordNet for Case-based Retrieval of UML Models". In: *AI Communications* 17.1 (2004), pp. 13–23 (cited on pages 8, 117).

[Góm+15] A. Gómez et al. "Map-Based Transparent Persistence for Very Large Models". In: *Fundamental Approaches to Software Engineering*. Ed. by A. Egyed and I. Schaefer. Vol. 9033. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 19–34. ISBN: 978-3-662-46674-2. DOI: 10.1007/978-3-662-46675-9_2 (cited on page 67).

[Gör+13] R. Görke et al. "Identifikation von Clustern in Graphen". In: *Informatik-Spektrum* 36.2 (2013), pp. 144–152. ISSN: 0170-6012. DOI: 10.1007/s00287-013-0685-0 (cited on pages 76, 82, 90).

[Gos+13] J. Gosling et al. *The Java Language Specification: Java SE 7 Edition*. Java SE 7 edition. Oracle America Inc., 2013. ISBN: 9780133260229. URL: https://docs.oracle.com/javase/specs/ (cited on pages 13, 30).

[GPC02] M. Genero, M. Piattini, and C. Calero. “Empirical Validation of Class Diagram Metrics”. In: *International Symposium in Empirical Software Engineering*. Vol. 7. Empirical Software Engineering. Kluwer Academic Publishers, 2002, pp. 195–203. DOI: 10.1109/ISESE.2002.1166940 (cited on page 104).

[GPC05] M. Genero, M. Piattini, and C. Calero. “A Survey of Metrics for UML Class Diagrams”. In: *The Journal of Object Technology* 4.9 (2005), pp. 59–92. ISSN: 1660-1769. DOI: 10.5381/jot.2005.4.9.a1 (cited on pages 4, 9, 79, 104, 117).

[GR03] A. Gerber and K. Raymond. “MOF to EMF: There and Back Again”. In: *The 2003 OOPSLA workshop*. Ed. by M. G. Burke. 2003, pp. 60–64. DOI: 10.1145/965660.965673 (cited on page 34).

[Gru93] T. R. Gruber. “A Translation Approach to Portable Ontology Specifications”. In: *Knowledge Acquisition* 5.2 (1993), pp. 199–220. ISSN: 10428143. DOI: 10.1006/knac.1993.1008 (cited on pages 68, 259).

[GT15] C. Gormley and Z. Tong. *Elasticsearch The Definitive Guide: A Distributed Real-Time Search and Analytics Engine*. O’Reilly and Associates, 2015. ISBN: 1449358543 (cited on pages 30, 67, 154).

[GVR02] R. L. Glass, I. Vessey, and V. Ramesh. “Research in software engineering: an analysis of the literature”. In: *Information and Software Technology* 44.8 (2002), pp. 491–506. ISSN: 09505849. DOI: 10.1016/S0950-5849(02)00049-6 (cited on pages 165, 166).

[Hei+09] F. Heidenreich et al. “Derivation and Refinement of Textual Syntax for Models”. In: *Model Driven Architecture - Foundations and Applications*. Ed. by R. F. Paige, A. Hartman, and A. Rensink. Vol. 5562. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 114–129. ISBN: 978-3-642-02673-7. DOI: 10.1007/978-3-642-02674-4_9 (cited on page 189).

[Hei12a] L. Heinemann. “Effective and Efficient Reuse with Software Libraries”. PhD thesis. Technische Universität München, 2012 (cited on pages 7, 8, 20, 22, 23, 148).

[Hei12b] L. Heinemann. “Facilitating Reuse in Model-Based Development with Context-Dependent Model Element Recommendations”. In: *Proceedints of the 3rd International Workshop on Recommendation Systems for Software Engineering (RSSE)*. ACM, 2012, pp. 16–20. DOI: 10.1109/RSSE.2012.6233402 (cited on pages 7, 23, 148).

[Hem+13] H. Hemmati et al. “The MSR Cookbook: Mining a Decade of Research”. In: *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories (MSR 2013)*. IEEE, 2013, pp. 343–352. ISBN: 978-1-4673-2936-1. DOI: 10.1109/MSR.2013.6624048 (cited on page 73).

[Her+06] H. Herre et al. “General Formal Ontology (GFO) - A Foundational Ontology Integrating Objects and Processes [Version 1.0]”. In: *Technical Report University of Leipzig* 8 (2006) (cited on page 68).

[Her11] M. Herrmannsdörfer. “Evolutionary Metamodeling”. PhD Thesis. München: Technische Universität München, 2011 (cited on pages 32, 38, 41, 46, 94, 113).

[Hes02] W. Hesse. “Ontologie(n)”. In: *Informatik-Spektrum* 25.6 (2002), pp. 477–480. ISSN: 0170-6012. DOI: 10.1007/s002870200265 (cited on page 68).

[Hes06a] W. Hesse. “Modelle - Janusköpfe der Software-Entwicklung - oder: Mit Janus von der A- zur S-Klasse”. In: *Modellierung 2006*. Ed. by R. Breu and H. C. Mayr. Vol. 82. GI-Edition Proceedings. Bonn: Gesellschaft für Informatik, 2006, pp. 99–113. ISBN: 3-88579-176-5. URL: http://dblp.uni-trier.de/db/conf/modellierung/modellierung2006.html%5C#Hesse06 (cited on page 18).

[Hes06b] W. Hesse. “More matters on (meta-)modelling: remarks on Thomas Kühne’s “matters””. In: *Software & Systems Modeling* 5.4 (2006), pp. 387–394. ISSN: 1619-1366. DOI: 10.1007/s10270-006-0033-9 (cited on page 17).

[HH10] M. Herrmannsdörfer and B. Hummel. “Library Concepts for Model Reuse”. In: *Electronic Notes in Theoretical Computer Science* 253.7 (2010), pp. 121–134. ISSN: 15710661. DOI: 10.1016/j.entcs.2010.08.036 (cited on page 72).

[Hil10] R. Hildebrandt. “Profile für bidirektionale Qualitätsmodelle in QMetric”. Diploma Thesis. Aachen, Germany: RWTH Aachen University, 2010 (cited on pages 112, 190).

[HJA08] O. Hummel, W. Janjic, and C. Atkinson. “Code Conjurer: Pulling Reusable Software out of Thin Air”. In: *IEEE Software* 25.5 (2008), pp. 45–52. ISSN: 0740-7459. DOI: 10.1109/MS.2008.110 (cited on page 144).

[HKL08] T. Hornung, A. Koschmider, and G. Lausen. “Recommendation Based Process Modeling Support: Method and User Experience”. In: *Conceptual Modeling - ER 2008*. Ed. by Q. Li et al. Vol. 5231. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 265–278. ISBN: 978-3-540-87876-6. DOI: 10.1007/978-3-540-87877-3_20 (cited on page 147).

[HKP12] J. Han, M. Kamber, and J. Pei. *Data mining: Concepts and techniques*. 3rd ed. The Morgan Kaufmann series in data management systems. Waltham, Mass.: Morgan Kaufmann Publishers, 2012. ISBN: 978-0-12-381479-1 (cited on pages 9, 13, 19, 73).

[HLN09] V. Hoffmann, H. Lichter, and A. Nyen. “Processes and Practices for Quality Scientific Software Projects”. In: *International Workshop on Academic Software Development Tools WASDeTT-3*. 2009, pp. 95–108 (cited on pages 24, 170, 184).

[HM08a] H.-J. Happel and W. Maalej. “Potentials and Challenges of Recommendation Systems for Software Development”. In: *Proceedings of the International Workshop on Recommendation Systems for Software Engineering*. New York, NY: ACM, 2008, p. 11. ISBN: 978-1-60558-228-3. DOI: 10.1145/1454247.1454251 (cited on pages 144, 157).

[HM08b] W. Hesse and H. C. Mayr. “Modellierung in der Softwaretechnik: eine Bestandsaufnahme”. In: *Informatik-Spektrum* 31.5 (2008), pp. 377–393. ISSN: 0170-6012. DOI: 10.1007/s00287-008-0276-7 (cited on pages 7, 17, 18).

[Hof13] V. Hoffmann. *Rapid Prototyping in der Use-Case-zentrierten Anforderungsanalyse*. 1., Aufl. Vol. 15. Aachener Informatik Berichte Software Engineering. Herzogenrath: Shaker, 2013. ISBN: 978-3-8440-2291-9 (cited on pages 24, 170).

[Hu13] X. Hu. “Model Templates for Ecore Libraries”. Diploma Thesis. Aachen, Germany: RWTH Aachen University, 2013 (cited on pages 65, 190).

[HVW11] M. Herrmannsdörfer, S. D. Vermolen, and G. Wachsmuth. “An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models”. In: *Third International Conference Software Language Engineering (SLE 2010)*. Ed. by B. Malloy, S. Staab, and Brand, Mark van den. Vol. 6563. Lecture notes in computer science. Berlin: Springer, 2011, pp. 163–182. ISBN: 978-3-642-19439-9. DOI: 10.1007/978-3-642-19440-5_10 (cited on page 113).

[HW12] R. Holmes and R. J. Walker. “Systematizing Pragmatic Software Reuse”. In: *ACM Transactions on Software Engineering and Methodology* 21.4 (2012), pp. 1–44. ISSN: 1049331X. DOI: 10.1145/2377656.2377657 (cited on pages 22, 23).

[HWM06] R. Holmes, R. Walker, and G. Murphy. “Approximate Structural Context Matching: An Approach to Recommend Relevant Examples”. In: *IEEE Transactions on Software Engineering* 32.12 (2006), pp. 952–970. ISSN: 00985589. DOI: 10.1109/TSE.2006.117 (cited on page 145).

[IEE10] IEEE Computer Society. *IEEE Standard for Information Technology - System and Software Life Cycle Processes - Reuse Processes*. New York: Institute of Electrical and Electronics Engineers, 2010. ISBN: 9780738163574 (cited on pages 21, 31).

[II01] ISO and IEC. *ISO/IEC 9126-1:2001 Software engineering – Product quality*. Geneva, CH, 2001 (cited on page 172).

[II11] ISO and IEC. *ISO/IEC 25000 Software engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE*. Geneva, CH, 2011. URL: https://www.iso.org/obp/ui/?_escaped_fragment_=iso:std:35733:en%5C#!iso:std:35733:en (cited on pages 166, 169, 172, 174–176, 178, 179).

[III06] ISO, IEC, and IEEE. *ISO/IEC 14764:2006: Software Engineering – Software Life Cycle Processes – Maintenance*. 2nd ed. Vol. ISO/IEC 14764. International standard. Geneva [Switzerland]: ISO, 2006. ISBN: 0-7381-4960-8 (cited on page 94).

[III10] ISO, IEC, and IEEE. *ISO/IEC/IEEE 24765 Systems and software engineering – Vocabulary: Ingénierie des systèmes et du logiciel – Vocabulaire*. 1st ed., 2010. Vol. 24765. International standard. Geneva, Switzerland and New York: ISO/IEC and Institute of Electrical and Electronics Engineers, 2010. ISBN: 978-0-7381-6205-8 (cited on pages 16–18, 165).

[Jan+11] D. Jannach et al. *Recommender Systems: An introduction*. New York: Cambridge University Press, 2011. ISBN: 978-0521493369. DOI: 10.1017/CB09780511763113 (cited on pages 9, 13, 20, 191).

[Jan+13] W. Janjic et al. "An Unabridged Source Code Dataset for Research in Software Reuse". In: *Proceedings of the 10th IEEE Working Conference on Mining Software Repositories (MSR 2013)*. IEEE, 2013, pp. 339–342. ISBN: 978-1-4673-2936-1. DOI: 10.1109/MSR.2013.6624047 (cited on page 145).

[Jea05a] Jean-marie Favre. "Foundations of Meta-Pyramids: Languages vs. Metamodels: Episode II: Story of Thotus the Baboon". In: *Language Engineering for Model-Driven Software Development*. Ed. by Jean Bezivin and Reiko Heckel. Dagstuhl Seminar Proceedings. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. URL: http://drops.dagstuhl.de/opus/volltexte/2005/21 (cited on pages 17, 29).

[Jea05b] Jean-marie Favre. "Foundations of Model (Driven) (Reverse) Engineering: Models – Episode I: Stories of The Fidus Papyrus and of The Solarus". In: *Language Engineering for Model-Driven Software Development*. Ed. by Jean Bezivin and Reiko Heckel. Dagstuhl Seminar Proceedings. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. URL: http://drops.dagstuhl.de/opus/volltexte/2005/13 (cited on pages 17, 29).

[Jea05c] Jean-marie Favre. "Megamodeling and Etymology: A story of Words: from MED to MDE via MODEL in five milleniums". In: *Dagstuhl Seminar on Transformation Techniques in Software Engineering*. 2005 (cited on pages 17, 29).

[JF88] R. E. Johnson and B. Foote. “Designing Reusable Classes”. In: *Journal of object-oriented programming* 1.2 (1988), pp. 22–35 (cited on page 28).

[JFC04] S. R. Judson, R. B. France, and D. L. Carver. “Supporting Rigorous Evolution of UML Models”. In: *Proceedings of the Ninth IEEE International Conference on Engineering Complex Computer Systems 2004*. IEEE, 2004, pp. 128–137. DOI: 10.1109/ICECCS.2004.1310911 (cited on page 98).

[JGB11] C. Jeanneret, M. Glinz, and B. Baudry. “Estimating Footprints of Model Operations”. In: *ICSE 2011*. New York: ACM, 2011, pp. 601–610. ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985875 (cited on pages 88, 90).

[JHA14] W. Janjic, O. Hummel, and C. Atkinson. “Reuse-Oriented Code Recommendation Systems”. In: *Recommendation Systems in Software Engineering*. Ed. by M. P. Robillard et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 359–386. ISBN: 978-3-642-45134-8. DOI: 10.1007/978-3-642-45135-5_14 (cited on pages 2, 7, 20, 23, 48, 186).

[Kap+06] G. Kappel et al. “On Models and Ontologies - A Semantic Infrastructure Supporting Model Integration”. In: *Modellierung 2006*. Ed. by R. Breu and H. C. Mayr. Vol. 82. GI-Edition Proceedings. Bonn: Gesellschaft für Informatik, 2006, pp. 11–27. ISBN: 3-88579-176-5 (cited on page 69).

[KB02] H. Kim and C. Boldyreff. “Developing Software Metrics Applicable to UML Models”. In: *Object-Oriented Technology ECOOP 2002 Workshop Reader*. Ed. by G. Goos et al. Vol. 2548. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002 (cited on page 114).

[KC00] R. Krishnan and K. Chari. “Model Management: Survey, Future Research Directions and a Bibliography”. In: *The Interactive Transactions of OR/MS* 3.1 (2000). URL: https://www.informs.org/Pubs/ITORMS/Archive/Volume-3/Volume-3-No.-1-Krishnan-and-Chari (cited on page 72).

[KCM07] H. Kagdi, M. L. Collard, and J. I. Maletic. “A survey and taxonomy of approaches for mining software repositories in the context of software evolution”. In: *Journal of Software Maintenance and Evolution: Research and Practice* 19.2 (2007), pp. 77–131. ISSN: 1532-060X. DOI: 10.1002/smr.344 (cited on page 73).

[KD99] S.-K. Kim and C. David. “Formalizing the UML Class Diagram Using Object-Z”. In: *UML’99 — The Unified Modeling Language*. Ed. by G. Goos et al. Vol. 1723. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 83–98. ISBN: 978-3-540-66712-4. DOI: 10.1007/3-540-46852-8_7 (cited on page 43).

[KE14] A. Koshima and V. Englebert. “Collaborative Editing of EMF/Ecore Metamodels and Models - Conflict Detection, Reconciliation, and Merging in DiCoMEF”. In: *Modelsward 2014, Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development, Lisbon, Portugal, 7.-9. January 2014*. SCITEPRESS, 2014, pp. 55–66. ISBN: 978-989-758-007-9. DOI: 10.5220/0004709500550066 (cited on pages 32, 34, 37, 41).

[KH10] M. Kögel and J. Helming. “EMFStore: a Model Repository for EMF Models”. In: *ACM/IEEE 32nd International Conference on Software Engineering, 2010*. Ed. by J. Kramer. Piscataway, NJ: IEEE, 2010, p. 307. ISBN: 1605587192. DOI: 10.1145/1810295.1810364 (cited on page 69).

[KHO11] A. Koschmider, T. Hornung, and A. Oberweis. “Recommendation-based Editor for Business Process Modeling”. In: *Data & Knowledge Engineering* 70.6 (2011), pp. 483–503. ISSN: 0169023X. DOI: 10.1016/j.datak.2011.02.002 (cited on pages 147, 148).

[KL70] B. W. Kernighan and S. Lin. “An Efficient Heuristic Procedure for Partitioning Graphs”. In: *Bell System Technical Journal* 49.2 (1970), pp. 291–307. ISSN: 00058580. DOI: 10.1002/j.1538-7305.1970.tb01770.x (cited on pages 80, 88).

[KM05] M. Kersten and G. C. Murphy. “Mylar: a degree-of-interest model for IDEs”. In: *Proceedings of the 4th international conference on Aspect-oriented software development*. Ed. by M. Mezini. New York, NY: ACM, 2005, pp. 159–168. ISBN: 1-59593-042-6. DOI: 10.1145/1052898.1052912 (cited on pages 124, 144, 191).

[KM06] M. Kersten and G. C. Murphy. “Using Task Context to Improve Programmer Productivity”. In: *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Ed. by M. Young and P. Devanbu. New York, NY: ACM Press, 2006, p. 1. ISBN: 1-59593-468-5. DOI: 10.1145/1181775.1181777 (cited on pages 124, 144, 191).

[KM14] T. Kuschke and P. Mäder. “Pattern-based Auto-completion of UML Modeling Activities”. In: *Proceedings of the 29th ACMIEEE international conference on Automated software engineering*. Ed. by I. Crnkovic. [S.l.]: ACM, 2014, pp. 551–556. ISBN: 978-1-4503-3013-8. DOI: 10.1145/2642937.2642949 (cited on pages 122, 146).

[KMG11] P. Kelsen, Q. Ma, and C. Glodt. “Models within Models: Taming Model Complexity Using the Sub-model Lattice”. In: *Fundamental Approaches to Software Engineering*. Ed. by D. Giannakopoulou and F. Orejas. Vol. 6603. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 171–185. ISBN: 978-3-642-19810-6. DOI: 10.1007/978-3-642-19811-3_13 (cited on pages 92, 93).

[KMR13] T. Kuschke, P. Mäder, and P. Rempel. “Recommending Auto-completions for Software Modeling Activities”. In: *Model-Driven Engineering Languages and Systems*. Ed. by A. Moreira et al. Vol. 8107. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 170–186. ISBN: 978-3-642-41532-6. DOI: 10.1007/978-3-642-41533-3_11 (cited on pages 143, 146).

[KMS05] H. Kagdi, J. I. Maletic, and A. Sutton. “Context-Free Slicing of UML Class Models”. In: *ICSM 2005*. Los Alamitos, Calif.: IEEE Computer Society, 2005, pp. 635–638. ISBN: 0-7695-2368-4. DOI: 10.1109/ICSM.2005.34 (cited on page 90).

[KO10] A. Koschmider and A. Oberweis. “Designing Business Processes with a Recommendation-Based Editor”. In: *Handbook on Business Process Management 1*. Ed. by J. Vom Brocke and M. Rosemann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 299–312. ISBN: 978-3-642-00415-5. DOI: 10.1007/978-3-642-00416-2_14 (cited on page 148).

[Kög11] M. Kögel. *Operation-based Model Evolution*. 1. Auflage. München: Verlag Dr. Hut, 2011. ISBN: 978-3843900812 (cited on pages 32, 41, 46, 48, 69, 94, 174).

[Kon11] Konrad Voigt. “Structural Graph-based Metamodel Matching”. PhD Thesis. Dresden: TU Dresden, 2011. URL: http://d-nb.info/1067729755/34 (cited on page 92).

[Kos07] A. Koschmider. *Ähnlichkeitsbasierte Modellierungsunterstützung für Geschäftsprozesse*. Karlsruhe: Univ.-Verl. Karlsruhe, 2007. ISBN: 3866441886 (cited on page 148).

[KPP09] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. “The Grand Challenge of Scalability for Model Driven Engineering”. In: *Models in Software Engineering*. Ed. by M. R. V. Chaudron. Vol. 5421. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 48–53. ISBN: 978-3-642-01647-9. DOI: 10.1007/978-3-642-01648-6_5 (cited on page 23).

[KR01] F. Keienburg and A. Rausch. “Using XML/XMI for Tool Supported Evolution of UML Models”. In: *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*. IEEE, 2001, p. 10. ISBN: 0-7695-0981-9. DOI: 10.1109/HICSS.2001.927259 (cited on page 95).

[Kru04] P. Kruchten. *The Rational Unified Process: An Introduction*. 3rd ed. The Addison-Wesley object technology series. Boston: Addison-Wesley, 2004. ISBN: 9780321197702 (cited on page 6).

[Kru92] C. W. Krueger. “Software Reuse”. In: *ACM Computing Surveys* 24.2 (1992), pp. 131–183. ISSN: 03600300. DOI: 10.1145/130844.130856 (cited on pages 21, 22).

[Küh06a] T. Kühne. “Clarifying matters of (meta-) modeling: an author’s reply”. In: *Software & Systems Modeling* 5.4 (2006), pp. 395–401. ISSN: 1619-1366. DOI: 10.1007/s10270-006-0034-8 (cited on page 17).

[Küh06b] T. Kühne. “Matters of (meta-) modeling”. In: *Software & Systems Modeling* 5.4 (2006), pp. 369–385. ISSN: 1619-1366. DOI: 10.1007/s10270-006-0017-9 (cited on page 17).

[Kuh10] A. Kuhn. “On Recommending Meaningful Names in Source and UML”. In: *2nd International Workshop on Recommendation Systems for Software Engineering*. Ed. by R. Holmes et al. 2010, pp. 50–51. DOI: 10.1145/1808920.1808932 (cited on page 146).

[KW07] E. Kindler and R. Wagner. “Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios”. Paderborn, 2007. URL: http://www.cs.uni-paderborn.de/uploads/tx_sibibtex/tr-ri-07-284.pdf (cited on page 146).

[LA12] M. Lines and S. Ambler. *Disciplined Agile Delivery: A Practitioner’s Guide to Agile Software Delivery in the Enterprise*. [S.l.]: IBM Press, 2012. ISBN: 9780132810135 (cited on page 6).

[Lan06] C. Lange. “Improving the Quality of UML Models in Practice”. In: *28th International Conference on Software Engineering*. New York, N.Y: Association for Computing Machinery, 2006, pp. 993–996. ISBN: 1-59593-375-1. DOI: 10.1145/1134285.1134472 (cited on page 114).

[Leh80] M. M. Lehman. “Programs, Life Cycles, and Laws of Software Evolution”. In: *Proceedings of the IEEE* 68.9 (1980), pp. 1060–1076. DOI: 10.1109/PROC.1980.11805 (cited on pages 94, 95, 113).

[Ler00] B. S. Lerner. “A Model for Compound Type Changes Encountered in Schema Evolution”. In: *ACM Transactions on Database Systems* 25.1 (2000), pp. 83–127. ISSN: 03625915. DOI: 10.1145/352958.352983 (cited on page 45).

[Ler06] X. Leroy. *Polymorphic typing of an algorithmic language*. 2006. URL: https://hal.inria.fr/inria-00077018/ (cited on page 41).

[Lev+11] T. Levendovszky et al. “Model Evolution and Management”. In: *Model-Based Engineering of Embedded Real-Time Systems*. Ed. by H. Giese et al. Vol. 6100. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 241–270. ISBN: 978-3-642-16276-3. DOI: 10.1007/978-3-642-16277-0_9 (cited on pages 72, 94, 113, 114).

[LFR12] S. Lehnert, Q.-a. Farooq, and M. Riebisch. “A Taxonomy of Change Types and Its Application in Software Evolution”. In: *2012 19th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS)*. 2012, pp. 98–107. DOI: 10.1109/ECBS.2012.9 (cited on pages 108, 115).

[LFW12] D. Lucrédio, R. P. Fortes, and J. Whittle. “MOOGLE: a metamodel-based model search engine”. In: *Software & Systems Modeling* 11.2 (2012), pp. 183–208. ISSN: 1619-1366. DOI: 10.1007/s10270-010-0167-7 (cited on pages 2, 70, 158).

[LGS11] P. Lops, M. d. Gemmis, and G. Semeraro. “Content-based Recommender Systems: State of the Art and Trends”. In: *Recommender Systems Handbook*. Ed. by F. Ricci et al. Boston, MA: Springer US, 2011, pp. 73–105. ISBN: 978-0387858197. DOI: 10.1007/978-0-387-85820-3_3 (cited on pages 20, 119, 125).

[LK10] K. Lano and S. Kolahdouz-Rahimi. “Slicing of UML Models Using Model Transformations”. In: *Model Driven Engineering Languages and Systems*. Ed. by D. C. Petriu, N. Rouquette, and Ø. Haugen. Vol. 6395. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 228–242. ISBN: 978-3-642-16128-5. DOI: 10.1007/978-3-642-16129-2_17 (cited on page 90).

[LK11] K. Lano and S. Kolahdouz-Rahimi. “Slicing Techniques for UML Models”. In: *The Journal of Object Technology* 10 (2011), 11:1. ISSN: 1660-1769. DOI: 10.5381/jot.2011.10.1.a11 (cited on page 91).

[LL10] J. Ludewig and H. Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. 2., überarb., aktualisierte und erg. Aufl. Heidelberg: dpunkt-Verl, 2010. ISBN: 9783898646628 (cited on pages 20–22, 166, 169, 192).

[LM08] J. T. Lallchandani and R. Mall. “Slicing UML Architectural Models”. In: *ACM SIGSOFT Software Engineering Notes* 33.3 (2008), p. 1. ISSN: 01635948. DOI: 10.1145/1360602.1360611 (cited on pages 76, 91).

[LM09] J. T. Lallchandani and R. Mall. “Static Slicing of UML Architectural Models”. In: *The Journal of Object Technology* 8.1 (2009), pp. 159–188. ISSN: 1660-1769. DOI: 10.5381/jot.2009.8.1.a2 (cited on pages 76, 91).

[LM12] J. Loeliger and M. McCullough. *Version Control with Git*. Second edition. O’Reilly and Associates, 2012. ISBN: 978-1-449-31638-9 (cited on pages 30, 153).

[LP79] R. G. Lanergan and B. A. Poynton. “Reusable Code: The Application Development Technique of the Future”. In: *Proceedings of the IBM SHARE GUIDE Software Symposium*. IBM, 1979 (cited on page 21).

[LR07] S. Limam Mansar and H. A. Reijers. “Best practices in business process redesign: use and impact”. In: *Business Process Management Journal* 13.2 (2007), pp. 193–213. ISSN: 1463-7154. DOI: 10.1108/14637150710740455 (cited on pages 21, 48).

[LS80] B. P. Lientz and E. B. Swanson. *Software Maintenance Management: A Study of the Maintenance of Computing Application Software in 487 Data Processing Organizations*. Reading MA: Addison-Wesley, 1980. ISBN: 0201042053 (cited on page 94).

[LSS94] O. I. Lindland, G. Sindre, and A. Solvberg. "Understanding Quality in Conceptual Modeling". In: *IEEE Software* 11.2 (1994), pp. 42–49. ISSN: 0740-7459. DOI: 10.1109/52.268955 (cited on pages 101, 104, 114).

[LSW87] M. Lenz, H. A. Schmid, and P. F. Wolf. "Software Reuse through Building Blocks". In: *IEEE Software* 4.4 (1987), pp. 34–42. ISSN: 0740-7459. DOI: 10.1109/MS.1987.231062 (cited on page 22).

[LSY03] G. Linden, B. Smith, and J. York. "Amazon.com Recommendations: Item-to-Item Collaborative Filtering". In: *IEEE Internet Computing* 7.1 (2003), pp. 76–80. ISSN: 1089-7801. DOI: 10.1109/MIC.2003.1167344 (cited on pages 2, 20, 117, 144).

[Luc97] C. Lucas. "Documenting Reuse and Evolution with Reuse Contracts". Ph.D. Bussels: Vrije Universiteit Brussel, 1997 (cited on page 23).

[Lud03] J. Ludewig. "Models in software engineering - an introduction". In: *Software and Systems Modeling* 2.1 (2003), pp. 5–14. ISSN: 1619-1366. DOI: 10.1007/s10270-003-0020-3 (cited on page 17).

[LV02] J. LARROSA and G. VALIENTE. "Constraint satisfaction algorithms for graph pattern matching". In: *Mathematical Structures in Computer Science* 12.04 (2002). ISSN: 0960-1295. DOI: 10.1017/S0960129501003577 (cited on page 82).

[Lv92] E. Lippe and N. van Oosterom. "Operation-based Merging". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 17. 1992, pp. 78–87 (cited on pages 41, 44).

[LWC07a] C. Lange, M. A. Wijns, and M. Chaudron. "A Visualization Framework for Task-Oriented Modeling Using UML". In: *40th Hawaii international conference on system sciences*. Piscataway, N.J.: IEEE, 2007, 289a. ISBN: 0-7695-2755-8. DOI: 10.1109/HICSS.2007.44 (cited on page 114).

[LWC07b] C. Lange, M. A. Wijns, and M. Chaudron. "MetricViewEvolution: UML-based Views for Monitoring Model Evolution and Quality". In: *11th European Conference on Software Maintenance and Reengineering*. Ed. by R. Krikhaar. Los Alamitos, Calif. [u.a.]: IEEE Computer Society, 2007, pp. 327–328. ISBN: 0-7695-2802-3. DOI: 10.1109/CSMR.2007.32 (cited on page 114).

[Mac92] B. Macchini. "Reusing Software with ESTRO (Evolving Software Repository)". In: *Proceedings of the 4th International Conference on Software Engineering and Knowledge Engineering*. IEEE, 1992, pp. 150–157. DOI: 10.1109/SEKE.1992.227935 (cited on page 68).

[Mar+15] S. Martinez-Fernandez et al. “Aggregating Empirical Evidence about the Benefits and Drawbacks of Software Reference Architectures”. In: *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, 2015, pp. 1–10. DOI: 10.1109/ESEM.2015.7321184 (cited on pages 183, 192).

[Mar01] R. Marinescu. “Detecting Design Flaws via Metrics in Object-Oriented Systems”. In: *Proceedings, 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems*. Ed. by Q. Li. Los Alamitos, Calif.: IEEE Computer Society, 2001, pp. 173–182. ISBN: 0-7695-1251-8. DOI: 10.1109/TOOLS.2001.941671 (cited on page 79).

[Mar83] J. Martin. *Managing the Data Base Environment*. Englewood Cliffs, N.J.: Prentice-Hall, 1983. ISBN: 0-135-50582-8 (cited on page 38).

[Mas+05] Masateru Tsunoda et al. “Javawock: A Java Class Recommender System Based on Collaborative Filtering”. In: *The 17th International conference on software engineering & knowledge engineering*. Ed. by W. Chu, N. Juzgado, and E. Wong. Skokie, Ill.: Knowledge Systems Institute Graduate School, 2005, pp. 491–497. ISBN: 1-891706-16-0 (cited on page 117).

[Mat10] Matt Might. *The illustrated guide to a Ph.D.* 2010. URL: http://matt.might.net/articles/phd-school-in-pictures/ (visited on 11/01/2010) (cited on page 182).

[Mat84] Y. Matsumoto. “Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels”. In: *IEEE Transactions on Software Engineering* SE-10.5 (1984), pp. 502–513. ISSN: 00985589. DOI: 10.1109/TSE.1984.5010274 (cited on page 21).

[MBC09] M. Monperrus, A. Beugnard, and J. Champeau. “A Definition of “Abstraction Level” for Metamodels”. In: *Proceedings of the 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS)*. 2009, pp. 315–320. DOI: 10.1109/ECBS.2009.41 (cited on pages 17, 32).

[MBG09] A. Mougenot, X. Blanc, and M.-P. Gervais. “D-Praxis: A Peer-to-Peer Collaborative Model Editing Framework”. In: *Distributed Applications and Interoperable Systems*. Ed. by T. Senivongse and R. Oliveira. Vol. 5523. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 16–29. ISBN: 978-3-642-02163-3. DOI: 10.1007/978-3-642-02164-0_2 (cited on page 42).

[McC01] “Introduction to IEEE Std. 1517 - Software Reuse Processes”. In: *Software Reuse*. Ed. by C. McClure. IEEE, 2001, pp. 3–16. ISBN: 9781118156681. DOI: 10.1109/9781118156681.ch1 (cited on page 21).

[McI68] M. D. McIlroy. “Mass Produced Software Components”. In: *Report on a conference sponsored by the NATO Science Committee*. Ed. by P. Naur and B. Randell. Brussels: Scientific Affairs Division, NATO, 1968, pp. 138–155 (cited on page 21).

[MD00] T. Mens and T. D’Hondt. “Automating Support for Software Evolution in UML”. In: *Automated Software Engineering* 7.1 (2000), pp. 39–59. ISSN: 09288910. DOI: 10.1023/A:1008765200695 (cited on pages 108, 115).

[MD08] T. Mens and S. Demeyer, eds. *Software Evolution: History and Challenges of Software Evolution*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. ISBN: 978-3-540-76439-7. DOI: 10.1007/978-3-540-76440-3 (cited on page 13).

[MD09] P. Mohagheghi and V. Dehlen. “Existing Model Metrics and Relations to Model Quality”. In: *ICSE Workshop on Software Quality, 2009, WOSQ’09*. Piscataway, NJ: IEEE, 2009, pp. 39–45. ISBN: 1424437431 (cited on page 114).

[Mel04] S. Melnik. *Generic Model Management: Concepts and Algorithms*. Vol. 2967. Lecture notes in computer science. Berlin and London: Springer, 2004. ISBN: 9783540219804 (cited on page 71).

[Men02] T. Mens. “A State-of-the-Art Survey on Software Merging”. In: *IEEE Transactions on Software Engineering* 28.5 (2002), pp. 449–462. ISSN: 00985589. DOI: 10.1109/TSE.2002.1000449 (cited on page 44).

[Men14] T. Menzies. “Data Mining”. In: *Recommendation Systems in Software Engineering*. Ed. by M. P. Robillard et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 39–75. ISBN: 978-3-642-45134-8. DOI: 10.1007/978-3-642-45135-5_3 (cited on pages 19, 20, 132).

[Mes10] G. Meszaros. *xUnit Test Patterns: Refactoring Test Code*. 4. print. The Addison-Wesley singature series. Upper Saddle River, NJ: Addison-Wesley, 2010. ISBN: 9780131495050. URL: http://xunitpatterns.com/ (cited on page 160).

[MF67] A. Mehrabian and S. R. Ferris. “Inference of Attitudes from Nonverbal Communication in Two Channels”. In: *Journal of Consulting Psychology* 31.3 (1967), pp. 248–252. ISSN: 0095-8891. DOI: 10.1037/h0024648 (cited on page 2).

[MFJ05] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. “Weaving Executability into Object-Oriented Meta-languages”. In: *Model driven engineering languages and systems*. Ed. by L. Briand. Vol. 3713. Lecture notes in computer science. Berlin: Springer, 2005, pp. 264–278. ISBN: 9783540290100. DOI: 10.1007/11557432_19 (cited on page 91).

[MFP06] N. H. Madhavji, Fernández Ramil, Juan Carlos, and D. E. Perry. *Software Evolution and Feedback: Theory and practice*. Chichester: Wiley, 2006. ISBN: 0470871806 (cited on pages 94, 98).

[MFR14] W. Maalej, T. Fritz, and R. Robbes. “Collecting and Processing Interaction Data for Recommendation Systems”. In: *Recommendation Systems in Software Engineering*. Ed. by M. P. Robillard et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 173–197. ISBN: 978-3-642-45134-8. DOI: 10.1007/978-3-642-45135-5_7 (cited on pages 125, 191).

[Mic+15] Michael Fellmann et al. “Requirements Catalog for Business Process Modeling Recommender Systems”. In: *Proceedings der 12. Internationale Tagung Wirtschaftsinformatik*. Merkur-Verlag, 2015, pp. 393–407 (cited on pages 2, 7, 147, 192).

[Mil+02] H. Mili et al. *Reuse based software engineering: Techniques, organization, and measurement*. A Wiley-Interscience publication. New York, NY [u.a.]: Wiley, 2002. ISBN: 978-0471398196 (cited on pages 21–23, 65, 66).

[Mil78] R. Milner. “A Theory of Type Polymorphism in Programming”. In: *Journal of Computer and System Sciences* 17.3 (1978), pp. 348–375. ISSN: 00220000. DOI: 10.1016/0022-0000(78)90014-4 (cited on page 41).

[MJM12] E. Murphy-Hill, R. Jiresal, and G. C. Murphy. “Improving Software Developers’ Fluency by Recommending Development Environment Commands”. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE-20)*. Ed. by W. Tracz. 2012, p. 1. ISBN: 978-1-4503-1614-9. DOI: 10.1145/2393596.2393645 (cited on page 148).

[MKG15] Q. Ma, P. Kelsen, and C. Glodt. “A generic model decomposition technique and its application to the Eclipse modeling framework”. In: *Software & Systems Modeling* 14.2 (2015), pp. 921–952. ISSN: 1619-1366. DOI: 10.1007/s10270-013-0348-2 (cited on pages 90, 93).

[MLA04] V. Maxville, C. Lam, and J. Armarego. “Learning to Select Software Components”. In: *Proceedings of the Sixteenth International Conference on Software Engineering and Knowledge Engineering*. Ed. by F. Maurer and G. Ruhe. Skokie, Ill.: Knowledge Systems Institute, 2004, pp. 421–426. ISBN: 1-891706-14-4 (cited on page 117).

[MLS98] T. Mens, C. Lucas, and P. Steyaert. “Supporting Disciplined Reuse and Evolution of UML Models”. In: *UML ’98 : beyond the notation*. Ed. by J. Bézivin and P.-A. Muller. Vol. 1618. Lecture notes in computer science. Berlin and London: Springer-Verlag, 1998, pp. 378–392. ISBN: 978-3-540-66252-5. DOI: 10.1007/978-3-540-48480-6_29 (cited on page 23).

[MM09] S. Mazanek and M. Minas. “Business Process Models as a Showcase for Syntax-Based Assistance in Diagram Editors”. In: *Model Driven Engineering Languages and Systems*. Ed. by A. Schürr and B. Selic. Vol. 5795. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 322–336. ISBN: 978-3-642-04424-3. DOI: 10.1007/978-3-642-04425-0_24 (cited on page 147).

[MM14] E. Murphy-Hill and G. C. Murphy. “Recommendation Delivery”. In: *Recommendation Systems in Software Engineering*. Ed. by M. P. Robillard et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 223–242. ISBN: 978-3-642-45134-8. DOI: 10.1007/978-3-642-45135-5_9 (cited on pages 122, 186).

[MMA14] N. Murakami, H. Masuhara, and T. Aotani. “Code Recommendation Based on a Degree-of-Interest Model”. In: *4th International Workshop on Recommendation Systems for Software Engineering (RSSE 2014)*. Ed. by R. Holmes, W. Janjic, and W. Maalej. 2014, pp. 28–29. ISBN: 9781450328456. DOI: 10.1145/2593822.2593828 (cited on pages 144, 191).

[MMM08a] S. Mazanek, S. Maier, and M. Minas. “An Algorithm for Hypergraph Completion According to Hyperedge Replacement Grammars”. In: *Graph Transformations*. Ed. by H. Ehrig et al. Vol. 5214. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 39–53. ISBN: 978-3-540-87404-1. DOI: 10.1007/978-3-540-87405-8_4 (cited on page 146).

[MMM08b] S. Mazanek, S. Maier, and M. Minas. “Auto-completion for Diagram Editors based on Graph Grammars”. In: *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2008, pp. 242–245. DOI: 10.1109/VLHCC.2008.4639094 (cited on pages 146, 147).

[MMM98] A. Mili, R. Mili, and R. T. Mittermeir. “A survey of software reuse libraries”. In: *Annals of Software Engineering* 5 (1998), pp. 349–414. ISSN: 10227091. DOI: 10.1023/A:1018964121953 (cited on pages 22, 65, 68).

[Moh+09] N. Moha et al. “Generic Model Refactorings”. In: *Model Driven Engineering Languages and Systems*. Ed. by A. Schürr and B. Selic. Vol. 5795. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 628–643. ISBN: 978-3-642-04424-3. DOI: 10.1007/978-3-642-04425-0_50 (cited on page 184).

[Moo05] D. L. Moody. “Theoretical and practical issues in evaluating the quality of conceptual models: current state and future directions”. In: *Data & Knowledge Engineering* 55.3 (2005), pp. 243–276. ISSN: 0169023X. DOI: 10.1016/j.datak.2004.12.005 (cited on pages 4, 8, 23, 101, 103, 104, 114, 193).

[Moo09] D. L. Moody. “The “Physics” of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering”. In: *IEEE Transactions on Software Engineering* 35.6 (2009), pp. 756–779. ISSN: 00985589. DOI: 10.1109/TSE.2009.67 (cited on page 98).

[MP06] J. A. McQuillan and J. F. Power. “On the Application of Software Metrics to UML Models”. In: *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems*. Ed. by T. Kühne. Vol. 4364. Lecture notes in computer science. ACM, 2006, pp. 217–226.

ISBN: 978-3-540-69488-5. DOI: 10.1007/978-3-540-69489-2_27 (cited on page 114).

[MRS08] C. D. Manning, P. Raghavan, and H. Schütze. *An Introduction to Information Retrieval*. New York: Cambridge University Press, 2008. ISBN: 0521865719 (cited on pages 19, 57, 93, 126).

[MTM09] Mohammad Munshi Shahin Shah, T.L. McCluskey, and Margaret M. West. "A Study of Synthesizing Artificial Intelligence (AI) Planning Domain Models by Using Object Constraints". In: *Proceedings of Computing and Engineering Annual Researchers' Conference 2009: CEARC09*. Ed. by Gary Lucas and Zhijie Xu. Huddersfield: University of Huddersfield, 2009, pp. 64–69. URL: http://eprints.hud.ac.uk/6863/ (cited on page 48).

[Mul+12] P.-A. Muller et al. "Modeling modeling modeling". In: *Software & Systems Modeling* 11.3 (2012), pp. 347–359. ISSN: 1619-1366. DOI: 10.1007/s10270-010-0172-x (cited on page 17).

[Mul12] D. Mularski. "Indexing Ecore Libraries". Bachelor Thesis. Aachen, Germany: RWTH Aachen University, 2012 (cited on pages 56, 153).

[Mur+07] L. G. Murta et al. "Odyssey-SCM: An integrated software configuration management infrastructure for UML models". In: *Science of Computer Programming* 65.3 (2007), pp. 249–274. ISSN: 01676423. DOI: 10.1016/j.scico.2006.05.011 (cited on page 115).

[Mur12] K. P. Murphy. *Machine Learning: A Probabilistic Perspective*. Adaptive computation and machine learning series. Cambridge, Mass.: MIT Press, 2012. ISBN: 978-0-262-01802-9 (cited on page 13).

[Mus+14] G. Mussbacher et al. "The Relevance of Model-Driven Engineering Thirty Years from Now". In: *Model-Driven Engineering Languages and Systems*. Ed. by J. Dingel et al. Vol. 8767. Lecture notes in computer science. Cham: Springer International Publishing, 2014, pp. 183–200. ISBN: 9783319116-525. DOI: 10.1007/978-3-319-11653-2_12 (cited on pages 192, 193).

[MvS03] T. Mens, van der Straeten, Ragnhild, and J. Simmonds. "Maintaining Consistency between UML Models with Description Logic Tools". In: *ECOOP Workshop on Object-Oriented Reengineering*. Vol. 3031. Lecture notes in computer science. 2003 (cited on page 115).

[MvS05] T. Mens, van der Straeten, Ragnhild, and J. Simmonds. "A Framework for Managing Consistency of Evolving UML Models". In: *Software Evolution with UML and XML* (2005), pp. 1–31 (cited on pages 108, 115).

[MW67] A. Mehrabian and M. Wiener. "Decoding of Inconsistend Communications". In: *Journal of Personality and Social Psychology* 6.1 (1967), pp. 109–114. ISSN: 0022-3514. DOI: 10.1037/h0024532 (cited on page 2).

[Nag13] R. Nagarajan. “Conceptual Foundations for Ecore Recommender Strategies”. Master Thesis. Aachen, Germany: RWTH Aachen University, 2013 (cited on page 141).

[Nei84] J. M. Neighbors. “The Draco Approach to Constructing Software from Reusable Components”. In: *IEEE Transactions on Software Engineering* SE-10.5 (1984), pp. 564–574. ISSN: 00985589. DOI: 10.1109/TSE.1984.5010280 (cited on page 22).

[NG04] M. Newman and M. Girvan. “Finding and evaluating community structure in networks”. In: *Physical Review E* 69.2 (2004). ISSN: 1539-3755. DOI: 10.1103/PhysRevE.69.026113 (cited on pages 78, 88).

[Nim13] J. L. Nimpa. “Simulation Environments for Ecore Library Recommenders”. Bachelor Thesis. Aachen, Germany: RWTH Aachen University, 2013 (cited on page 160).

[Obe14a] D. Oberle. “How Ontologies Benefit Enterprise Applications”. In: *Semantic Web* 5.6 (2014). URL: http://iospress.metapress.com/content/k16n012507037044/ (cited on page 68).

[Obe14b] D. Oberle. “Ontologies and Reasoning in Enterprise Service Ecosystems”. In: *Informatik-Spektrum* 37.4 (2014), pp. 318–328. ISSN: 0170-6012. DOI: 10.1007/s00287-014-0785-5 (cited on pages 48, 68).

[Obj02] Object Management Group Inc. *Meta Object Facility (MOF™) Core Specification: ISO/IEC 19502*. 2002. URL: http://www.omg.org/spec/MOF/1.4/ (visited on 05/01/2015) (cited on page 33).

[Obj11a] Object Management Group Inc. *Unified Modeling Language™ (UML): Inftastructure*. 2011. URL: http://www.omg.org/spec/UML/2.4.1/ (visited on 05/01/2015) (cited on pages 17, 33, 34).

[Obj11b] Object Management Group Inc. *Unified Modeling Language™ (UML): Superstructure*. 2011. URL: http://www.omg.org/spec/UML/2.4.1/ (visited on 05/01/2015) (cited on page 115).

[Obj14] Object Management Group Inc. *Meta Object Facility (MOF™) Core Specification*. 2014. URL: http://www.omg.org/spec/MOF/2.4.2 (visited on 05/01/2015) (cited on pages 16, 29, 33, 34, 37, 113).

[OH06] A. Oshima and A. Hogue. *Writing Academic English*. 4th Edition. White Plains NY: Pearson/Longman, 2006. ISBN: 978-0131523593 (cited on page 10).

[OSG14] OSGi Alliance. *OSGi Core Release 6*. 2014. URL: http://www.osgi.org/Download/Release6 (cited on page 13).

[Pal+12] F. Palma et al. “Recommendation System for Design Patterns in Software Development: An DPR overview”. In: *Proceedings of the Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*. 2012, pp. 1–5. DOI: 10.1109/RSSE.2012.6233399 (cited on page 146).

[Par94] D. L. Parnas. “Software Aging”. In: *16th International Conference on Software Engineering*. 1994, pp. 279–287. DOI: 10.1109/ICSE.1994.296790 (cited on page 110).

[PC85] D. L. Parnas and P. C. Clements. “A Rational Design Process: How and Why to Fake It”. In: *Formal Methods and Software Development*. Ed. by G. Goos et al. Vol. 186. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, pp. 80–100. ISBN: 978-3-540-15199-9. DOI: 10.1007/3-540-15199-0_6 (cited on pages 169, 170).

[PF87] R. Prieto-Diaz and P. Freeman. “Classifying Software for Reusability”. In: *IEEE Software* 4.1 (1987), pp. 6–16. ISSN: 0740-7459. DOI: 10.1109/MS.1987.229789 (cited on page 23).

[PFW95] J. Petro, M. E. Fotta, and D. B. Weisman. “Model-Based Reuse Repositories - Concepts and Experience”. In: *Proceedings of the 7th International Workshop on Computer-Aided Software Engineering*. Ed. by H. A. Müller and R. D. Norman. IEEE, 1995, pp. 60–69. ISBN: 0818670789. DOI: 10.1109/CASE.1995.465328 (cited on pages 22, 65).

[Pic+11] J. Picault et al. “How to Get the Recommender Out of the Lab?” In: *Recommender Systems Handbook*. Ed. by F. Ricci et al. Boston, MA: Springer US, 2011, pp. 333–365. ISBN: 978-0387858197. DOI: 10.1007/978-0-387-85820-3_10 (cited on page 174).

[Pic03] G. Pickert. *Einführung in Ontologien*. Berlin, 2003. URL: http://www.dbis.informatik.hu-berlin.de/dbisold/lehre/WS0203/SemWeb/artikel/2/Pickert_Ontologien_final.pdf (cited on page 68).

[Pic09] A. C. Pick. *Discourse and Function: A Framework of Sentence Structure*. 2009. URL: http://www.discourseandfunction.com (cited on page 10).

[PJ07] M. Pizka and E. Jurgens. “Automating Language Evolution”. In: *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE ’07)*. 2007, pp. 305–315. DOI: 10.1109/TASE.2007.13 (cited on page 46).

[Por80] M. F. Porter. “An algorithm for suffix stripping”. In: *Program: electronic library and information systems* 14.3 (1980), pp. 130–137. ISSN: 0033-0337. DOI: 10.1108/eb046814 (cited on pages 19, 57, 93, 126).

[Pou99] J. S. Poulin. “Reuse: Been There, Done That”. In: *Communications of the ACM* 42.5 (1999), pp. 98–100. ISSN: 00010782. DOI: 10.1145/301353.301440 (cited on pages 21, 23).

[Pre96] W. Pree. *Framework Patterns*. New York: SIGS Books & Multimedia, 1996. ISBN: 1884842542 (cited on page 158).

[Pri91] R. Prieto-Díaz. “Making Software Reuse Work: An Implementation Model”. In: *ACM SIGSOFT Software Engineering Notes* 16.3 (1991), pp. 61–68. ISSN: 01635948. DOI: 10.1145/127099.127107 (cited on page 21).

[PS87] D. J. Penney and J. Stein. “Class modification in the GemStone object-oriented DBMS”. In: *ACM SIGPLAN Notices* 22.12 (1987), pp. 111–117. ISSN: 03621340. DOI: 10.1145/38807.38817 (cited on page 45).

[Pu+11] P. Pu et al. “Usability Guidelines for Product Recommenders Based on Example Critiquing Research”. In: *Recommender Systems Handbook*. Ed. by F. Ricci et al. Boston, MA: Springer US, 2011, pp. 511–545. ISBN: 978-0387858197. DOI: 10.1007/978-0-387-85820-3_16 (cited on pages 122, 123).

[PvM15] J. Pinna Puissant, van der Straeten, Ragnhild, and T. Mens. “Resolving model inconsistencies using automated regression planning”. In: *Software & Systems Modeling* 14.1 (2015), pp. 461–481. ISSN: 1619-1366. DOI: 10.1007/s10270-013-0317-9 (cited on page 114).

[QT06] A. Queralt and E. Teniente. “Reasoning on UML Class Diagrams with OCL Constraints”. In: *Conceptual Modeling - ER 2006*. Ed. by D. W. Embley, A. Olivé, and S. Ram. Vol. 4215. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 497–512. ISBN: 978-3-540-47224-7. DOI: 10.1007/11901181_37 (cited on page 147).

[Rag11] R. Ragimov. “Technische Verknüpfungen in Ecore Modellbibliotheken”. Bachelor Thesis. Aachen, Germany: RWTH Aachen University, 2011 (cited on page 82).

[Raj14] V. Rajlich. “Software Evolution and Maintenance”. In: *Future of Software Engineering (FOSE 2014)*. Ed. by M. Dwyer and J. Herbsleb. Hyderabad, India, 2014, pp. 133–144. ISBN: 978-1-4503-2865-4. DOI: 10.1145/2593882.2593893 (cited on page 94).

[RD39] F. J. Roethlisberger and W. J. Dickson. *Management and the Worker: An account of a research program conducted by the Western Electric Company, Hawthorne Works, Chicago. With the assistance and collaboration of Harold A. Wright*. 1. print. Cambridge, Mass.: Harvard Univ. Press, 1939. ISBN: 0674546768 (cited on page 165).

[Re04] A. Ramirez and et al. *ArgoUML User Manual v0.16*. 2004. URL: http://argouml.tigris.org (cited on page 106).

[Rei13] R. Reiners. *An Evolving Pattern Library for Collaborative Project Documentation*. 1. Aufl. Vol. 2014,1. Fraunhofer Series in Information and Communication Technology. Aachen: Shaker, 2013. ISBN: 3844027971 (cited on page 190).

[Rei83] Reinhardt Grossmann. *The Categorical Structure of the World*. Binding, 1983 (cited on page 68).

[RF63] R. Rosenthal and K. L. Fode. “The effect of experimenter bias on the performance of the albino rat”. In: *Behavioral Science* 8.3 (1963), pp. 183–189. ISSN: 00057940. DOI: 10.1002/bs.3830080302 (cited on page 165).

[Ric+11] F. Ricci et al., eds. *Recommender Systems Handbook*. Boston, MA: Springer US, 2011. ISBN: 978-0387858197. DOI: 10.1007/978-0-387-85820-3 (cited on pages 2, 9, 13, 20).

[Rie96] A. J. Riel. *Object-Oriented Design Heuristics*. Reading, Mass.: Addison-Wesley Pub. Co., 1996. ISBN: 020163385X (cited on page 79).

[RN10] M. A. Rodriguez and P. Neubauer. "The Graph Traversal Pattern". In: *CoRR* (2010). URL: http://arxiv.org/abs/1004.1001v1 (cited on page 30).

[Rob+14] M. P. Robillard et al., eds. *Recommendation Systems in Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014. ISBN: 978-3-642-45134-8. DOI: 10.1007/978-3-642-45135-5 (cited on pages 122, 259).

[Rob09] R. Robbes. "On the Ealuation of Recommender Systems with Recorded Interactions". In: *ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation (SUITE)*. ICSE. ACM, 2009, pp. 45–48. ISBN: 978-1-4244-3740-5. DOI: 10.1109/SUITE.2009.5070021 (cited on pages 166, 184).

[ROD10] T. Rahmani, D. Oberle, and M. Dahms. "An Adjustable Transformation from OWL to Ecore". In: *Model Driven Engineering Languages and Systems*. Ed. by D. C. Petriu, N. Rouquette, and Ø. Haugen. Vol. 6395. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 243–257. ISBN: 978-3-642-16128-5. DOI: 10.1007/978-3-642-16129-2_18 (cited on pages 68, 71).

[Rom+07] J. R. Romero et al. "Formal and Tool Support for Model Driven Engineering with Maude". In: *The Journal of Object Technology* 6.9 (2007), p. 187. ISSN: 1660-1769. DOI: 10.5381/jot.2007.6.9.a10 (cited on pages 32, 38, 43).

[Ros+09] L. M. Rose et al. *An Analysis of Approaches to Model Migration*. 2009. URL: http://www.modse.fr/modsemccm09/lib/exe/fetch.php?id=proceedings%5C&cache=cache%5C&media=modse-mccm-preliminary-proceedings.pdf (cited on page 45).

[Ros12] K. H. Rosen. *Discrete Mathematics and Its Applications*. 7th ed. New York: McGraw-Hill, 2012. ISBN: 978-0-07-338309-5 (cited on pages 14–16).

[Rot+13] A. Roth et al. "Staged Evolution with Quality Gates for Model Libraries". In: *Proceedings of the 1st International Workshop on:(Document) Changes: modeling, detection, storage and visualization*. ACM, 2013, pp. 1–8 (cited on pages 9, 10, 95, 153).

[Rot12] A. Roth. "Quality Staged Model Evolution in Ecore Libraries". Master Thesis. Aachen, Germany: RWTH Aachen University, 2012 (cited on pages 106, 110, 112, 175).

[Rot13] A. Roth. *A Metrics Mapping and Sources*. 2013. URL: http://goo.gl/ruqFpi (cited on page 106).

[RRS11] F. Ricci, L. Rokach, and B. Shapira. “Introduction to Recommender Systems Handbook”. In: *Recommender Systems Handbook*. Ed. by F. Ricci et al. Boston, MA: Springer US, 2011, pp. 1–35. ISBN: 978-0387858197. DOI: 10.1007/978-0-387-85820-3_1 (cited on pages 20, 169).

[RSA13] J. Reimann, M. Seifert, and U. Aßmann. “On the reuse and recommendation of model refactoring specifications”. In: *Software & Systems Modeling* 12.3 (2013), pp. 579–596. ISSN: 1619-1366. DOI: 10.1007/s10270-012-0243-2 (cited on pages 143, 184).

[Run+12] P. Runeson et al. *Case Study Research in Software Engineering: Guidelines and Examples*. 1st ed. Hoboken, N.J.: Wiley, 2012. ISBN: 9781118104354 (cited on page 165).

[RW14] M. P. Robillard and R. J. Walker. “An Introduction to Recommendation Systems in Software Engineering”. In: *Recommendation Systems in Software Engineering*. Ed. by M. P. Robillard et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 1–11. ISBN: 978-3-642-45134-8. DOI: 10.1007/978-3-642-45135-5_1 (cited on pages 7, 20, 118, 119, 197).

[RWE13] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases*. First edition. Sebastopol, Calif.: O’Reilly Media, 2013. ISBN: 978-1-449-35626-2 (cited on pages 13, 30, 56, 84, 153, 168).

[RWZ10] M. Robillard, R. Walker, and T. Zimmermann. “Recommendation Systems for Software Engineering”. In: *IEEE Software* 27.4 (2010), pp. 80–86. ISSN: 0740-7459. DOI: 10.1109/MS.2009.161 (cited on pages 2, 119, 144).

[SA13] H. O. Salami and M. A. Ahmed. “UML Artifacts Reuse: State of the Art”. In: *International Journal of Soft Computing and Software Engineering* 3.3 (2013), pp. 115–122. DOI: 10.7321/jscse.v3.n3.19 (cited on pages 18, 23, 145).

[Saf14] J. E. Safra. *Encyclopaedia Britannica*. Chicago: Encyclopaedia Britannica, 2014. ISBN: 1-59339-292-3 (cited on pages 94, 95, 106).

[San78] E. Sandewall. “Programming in an Interactive Environment: the “Lisp” Experience”. In: *ACM Computing Surveys* 10.1 (1978), pp. 35–71. ISSN: 03600300. DOI: 10.1145/356715.356719 (cited on page 8).

[SBP07] S. Sen, B. Baudry, and D. Precup. “Partial Model Completion in Model Driven Engineering using Constraint Logic Programming”. In: *International Conference on Applications of Declarative Programming and Knowledge Management*. Würzburg, Germany, 2007. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.386.1873 (cited on page 145).

[SBV08] S. Sen, B. Baudry, and H. Vangheluwe. “Domain-Specific Model Editors with Model Completion”. In: *Models in Software Engineering*. Ed. by H. Giese. Vol. 5002. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 259–270. ISBN: 978-3-540-69069-6. DOI: 10.1007/978-3-540-69073-3_27 (cited on page 145).

[SBV10] S. Sen, B. Baudry, and H. Vangheluwe. “Towards Domain-specific Model Editors with Automatic Model Completion”. In: *SIMULATION* 86.2 (2010), pp. 109–126. ISSN: 0037-5497. DOI: 10.1177/0037549709340530 (cited on page 145).

[SCB11] B. Smyth, M. Coyle, and P. Briggs. “Communities, Collaboration, and Recommender Systems in Personalized Web Search”. In: *Recommender Systems Handbook*. Ed. by F. Ricci et al. Boston, MA: Springer US, 2011, pp. 579–614. ISBN: 978-0387858197. DOI: 10.1007/978-0-387-85820-3_18 (cited on page 117).

[Sch+10] M. Schmidt et al. “An Integrated Facet-Based Library for Arbitrary Software Components”. In: *Modelling Foundations and Applications*. Ed. by T. Kühne et al. Vol. 6138. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 261–276. ISBN: 978-3-642-13594-1. DOI: 10.1007/978-3-642-13595-8_21 (cited on pages 66, 71).

[Sch10] H. Schackmann. *Metrik-basierte Auswertung von Software-Entwicklungsarchiven zur Prozessbewertung*. Vol. Bd. 7. Aachener Informatik-Berichte, Software-Engineering. Aachen: Shaker, 2010. ISBN: 978-3-8322-9405-2 (cited on page 112).

[Sch13] D. Schiller. “Cockpit Support for Ecore Library Recommenders”. Bachelor Thesis. Aachen, Germany: RWTH Aachen University, 2013 (cited on page 158).

[Sch95] A. Schürr. “Specification of Graph Translators with Triple Graph Grammars”. In: *Graph-Theoretic Concepts in Computer Science*. Ed. by E. W. Mayr, G. Schmidt, and G. Tinhofer. Vol. 903. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 151–163. ISBN: 978-3-540-59071-2. DOI: 10.1007/3-540-59071-4_45 (cited on pages 32, 146).

[Sco13a] T. Scott. *How NOT to Store Passwords!* 2013. URL: https://youtu.be/8ZtInClXe1Q (cited on page 3).

[Sco13b] T. Scott. *The Problem with Time & Timezones*. YouTube, 2013. URL: https://youtu.be/-5wpm-gesOY (cited on page 3).

[Seg03] J. Segal. “The Nature of Evidence in Empirical Software Engineering”. In: *Eleventh Annual International Workshop on Software Technology and Engineering Practice*. IEEE, 2003, pp. 40–47. DOI: 10.1109/STEP.2003.33 (cited on page 165).

[Sel03] B. Selic. “The Pragmatics of Model-Driven Development”. In: *IEEE Software* 20.5 (2003), pp. 19–25. ISSN: 0740-7459. DOI: 10.1109/MS.2003.1231146 (cited on page 17).

[Sen+09] S. Sen et al. “Meta-model Pruning”. In: *Model Driven Engineering Languages and Systems*. Ed. by A. Schürr and B. Selic. Vol. 5795. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 32–46. ISBN: 978-3-642-04424-3. DOI: 10.1007/978-3-642-04425-0_4 (cited on pages 88, 91).

[Sew13] N. Sewing. “Mining Data for Ecore Libraries”. Master Thesis. Aachen, Germany: RWTH Aachen University, 2013 (cited on pages 74, 154, 175).

[SF13] P. J. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Upper Saddle River, NJ: Addison-Wesley, 2013. ISBN: 0-321-82662-0 (cited on page 13).

[SF94] J. M. Swales and C. B. Feak. *Academic Writing for Graduate Students: Essential Tasks and Skills*. English for specific purposes. Ann Arbor: University of Michigan Press, 1994. ISBN: 0472082639 (cited on page 10).

[SF97] M. Shroff and R. B. France. “Towards a Formalization of UML Class Structures in Z”. In: *Twenty-First Annual International Computer Software and Applications Conference (COMPSAC’97)*. 1997, pp. 646–651. DOI: 10.1109/CMPSAC.1997.625087 (cited on page 43).

[SFR13] W. Sun, R. B. France, and I. Ray. “Contract-Aware Slicing of UML Class Models”. In: *Model-Driven Engineering Languages and Systems*. Ed. by A. Moreira et al. Vol. 8107. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 724–739. ISBN: 978-3-642-41532-6. DOI: 10.1007/978-3-642-41533-3_44 (cited on pages 76, 91).

[SG11] G. Shani and A. Gunawardana. “Evaluating Recommendation Systems”. In: *Recommender Systems Handbook*. Ed. by F. Ricci et al. Boston, MA: Springer US, 2011, pp. 257–297. ISBN: 978-0387858197. DOI: 10.1007/978-0-387-85820-3_8 (cited on page 184).

[Sha+10] A. Shaikh et al. “Verification-Driven Slicing of UML/OCL Models”. In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. Ed. by C. Pecheur, J. Andrews, and E. Di Nitto. New York, N.Y.: Association for Computing Machinery, 2010, pp. 185–194. ISBN: 978-1-4503-0116-9. DOI: 10.1145/1858996.1859038 (cited on pages 91, 92).

[SHL75] N. C. Shu, B. C. Housel, and V. Y. Lum. “CONVERT: A High Level Translation Definition Language for Data Conversion”. In: *Communications of the ACM* 18.10 (1975), pp. 557–567. ISSN: 00010782. DOI: 10.1145/361020.361023 (cited on page 44).

[Shu+77] N. C. Shu et al. “EXPRESS: A Data EXtraction, Processing, and Restructuring System”. In: *ACM Transactions on Database Systems* 2.2 (1977), pp. 134–174. ISSN: 03625915. DOI: 10.1145/320544.320549 (cited on page 44).

[SHW98] R. C. Seacord, S. A. Hissam, and K. C. Wallnau. “AGORA: A Search Engine for Software Components”. In: *IEEE Internet Computing* 2.6 (1998), p. 62. ISSN: 1089-7801. DOI: 10.1109/4236.735988 (cited on page 68).

[SIK15] S. A. Safdar, M. Z. Iqbal, and M. U. Khan. “Empirical Evaluation of UML Modeling Tools – A Controlled Experiment”. In: *Modelling Foundations and Applications*. Ed. by G. Taentzer and F. Bordeleau. Vol. 9153. Lecture notes in computer science. Cham: Springer International Publishing, 2015, pp. 33–44. ISBN: 978-3-319-21150-3. DOI: 10.1007/978-3-319-21151-0_3 (cited on page 6).

[SKR99] J. B. Schafer, J. Konstan, and J. Riedi. “Recommender Systems in E-Commerce”. In: *Proceedings of the ACM Conference on Electronic Commerce (EC’99)*. Ed. by S. Feldman. New York: ACM Press, 1999, pp. 158–166. ISBN: 9781581131765. DOI: 10.1145/336992.337035 (cited on pages 2, 144).

[SM13] Santos, José Amancio M. and M. G. Mendonça. “Identifying strategies on god class detection in two controlled experiments”. In: *The 26th International Conference on Software Engineering and Knowledge Engineering*. Ed. by Marek Reformat. Knowledge Systems Institute Graduate School, 2013, pp. 244–249 (cited on page 79).

[Som11] I. Sommerville. *Software Engineering*. 9th edition. Boston: Pearson, 2011. ISBN: 978-0-13-703515-1 (cited on pages 3, 21, 22, 179).

[Spr15] Springer International Publishing AG. *SpringerLink*. Cham, CH, 2015. URL: http://link.springer.com/ (visited on 06/25/2015) (cited on pages 4, 5).

[Spr80] R. H. Sprague. “A Framework for the Development of Decision Support Systems”. In: *MIS Quarterly* 4.4 (1980), p. 1. ISSN: 02767783. DOI: 10.2307/248957 (cited on pages 20, 144, 192).

[SS11] E. I. Sparling and S. Sen. “Rating: How Difficult is It?” In: *Proceedings of the fifth ACM conference on Recommender systems*. Ed. by B. Mobasher. New York, NY: ACM, 2011, p. 149. ISBN: 978-1-4503-0683-6. DOI: 10.1145/2043932.2043961 (cited on page 196).

[SSC09] Q. Shao, P. Sun, and Y. Chen. “WISE: a Workflow Information Search Engine”. In: *International Conference on Data Engineering (ICDE)*. 2009, pp. 1491–1494. DOI: 10.1109/ICDE.2009.89 (cited on page 70).

[ST82a] B. Shneiderman and G. Thomas. “An Architecture for Automatic Relational Database System Conversion”. In: *ACM Transactions on Database Systems* 7.2 (1982), pp. 235–257. ISSN: 03625915. DOI: 10.1145/319702.319724 (cited on page 44).

[ST82b] B. Shneiderman and G. Thomas. “Automatic database system conversion: schema revision, data translation, and source-to-source program transformation”. In: *AFIPS Proceedings*. New York, N.Y.: ACM Press, 1982, pp. 579–587. ISBN: 0-88283-035-X. DOI: 10.1145/1500774.1500849 (cited on page 44).

[Sta73] H. Stachowiak. *Allgemeine Modelltheorie*. S.l: Vienna Springer, 1973. ISBN: 9783211811061 (cited on page 17).

[Sta84] T. A. Standish. “An Essay on Software Reuse”. In: *IEEE Transactions on Software Engineering* SE-10.5 (1984), pp. 494–497. ISSN: 00985589. DOI: 10.1109/TSE.1984.5010272 (cited on pages 21, 22).

[STC14] A. Said, D. Tikk, and P. Cremonesi. “Benchmarking”. In: *Recommendation Systems in Software Engineering*. Ed. by M. P. Robillard et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 275–300. ISBN: 978-3-642-45134-8. DOI: 10.1007/978-3-642-45135-5_11 (cited on page 191).

[Ste+08] D. Steinberg et al. *EMF - Eclipse Modeling Framework*. 2. ed., rev. and updated. The Eclipse series. Upper Saddle River, NJ and Munich [u.a.]: Addison Wesley, 2008. ISBN: 9780321331885 (cited on pages 13, 18, 28, 29, 32, 34–37, 42, 104, 189).

[Ste+96] P. Steyaert et al. “Reuse Contracts: Managing the Evolution of Reusable Assets”. In: *ACM SIGPLAN Notices* 31.10 (1996), pp. 268–285. ISSN: 03621340. DOI: 10.1145/236338.236363 (cited on page 44).

[Stö14] H. Störrle. “On the Impact of Layout Quality to Understanding UML Diagrams: Size Matters”. In: *Model-Driven Engineering Languages and Systems*. Ed. by J. Dingel et al. Vol. 8767. Lecture notes in computer science. Cham: Springer International Publishing, 2014, pp. 518–534. ISBN: 978-3-319-11652-5. DOI: 10.1007/978-3-319-11653-2_32 (cited on pages 89, 101).

[Str+13] D. Strüber et al. “Towards a Distributed Modeling Process Based on Composite Models”. In: *Fundamental Approaches to Software Engineering*. Ed. by V. Cortellessa and D. Varró. Vol. 7793. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 6–20. ISBN: 978-3-642-37056-4. DOI: 10.1007/978-3-642-37057-1_2 (cited on pages 73, 92).

[Str+14] D. Struber et al. “Splitting Models Using Information Retrieval and Model Crawling Techniques”. In: *Fundamental Approaches to Software Engineering*. Ed. by S. Gnesi and A. Rensink. Vol. 8411. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 47–62. ISBN: 978-3-642-54803-1. DOI: 10.1007/978-3-642-54804-8_4 (cited on pages 77, 78, 89, 92).

[Stu09] H. Stuckenschmidt. *Ontologien: Konzepte, Technologien und Anwendungen*. 1st ed. Informatik im Fokus. Dordrecht and New York: Springer, 2009. ISBN: 978-3540793304. DOI: 10.1007/978-3-642-05404-4 (cited on page 68).

[Su+01] H. Su et al. "XEM: Managing the Wvolution of XML Socuments". In: *IEEE 11th Workshop on Research Issues in Data Engineering*. 2001, pp. 103–110. DOI: 10.1109/RIDE.2001.916497 (cited on page 47).

[SU13] F. Steimann and B. Ulke. "Generic Model Assist". In: *Model-Driven Engineering Languages and Systems*. Ed. by A. Moreira et al. Vol. 8107. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 18–34. ISBN: 978-3-642-41532-6. DOI: 10.1007/978-3-642-41533-3_2 (cited on pages 122, 143, 146, 184).

[Swa76] E. B. Swanson. "The Dimensions of Maintenance". In: *ICSE '76 Proceedings of the 2nd International Conference on Software Engineering*. Ed. by R. T. Yeh and C. V. Ramamoorthy. IEEE, 1976, pp. 492–497 (cited on page 113).

[SWG09] Y. Sun, J. White, and J. Gray. "Model Transformation by Demonstration". In: *Model Driven Engineering Languages and Systems*. Ed. by A. Schürr and B. Selic. Vol. 5795. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 712–726. ISBN: 978-3-642-04424-3. DOI: 10.1007/978-3-642-04425-0_58 (cited on page 114).

[Tai79] K.-C. Tai. "The Tree-to-Tree Correction Problem". In: *Journal of the ACM* 26.3 (1979), pp. 422–433. ISSN: 00045411. DOI: 10.1145/322139.322143 (cited on page 42).

[Tay+11] R. N. Taylor et al. "MT-Scribe: An End-User Approach to Automate Software Model Evolution". In: *ICSE 2011*. New York: ACM, 2011, pp. 980–982. ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985966 (cited on page 114).

[TDM10] R. N. Taylor, E. M. Dashofy, and N. Medvidović. *Software Architecture: Foundations, Theory, and Practice*. Hoboken, N.J.: Wiley, 2010. ISBN: 978-0470167748 (cited on pages 21, 48, 183, 192).

[TGL13] N. V. Tran, A. Ganser, and H. Lichter. "Multi Back-Ends for a Model Library Abstraction Layer". In: *Computational Science and Its Applications – ICCSA 2013*. Ed. by B. Murgante et al. Vol. 7973. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 160–174. ISBN: 978-3-642-39645-8. DOI: 10.1007/978-3-642-39646-5_12 (cited on pages 49, 153, 154, 168).

[Tic+95] W. F. Tichy et al. "Experimental Evaluation in Computer Science: A Quantitative Study". In: *Journal of Systems and Software* 28.1 (1995), pp. 9–18. ISSN: 01641212. DOI: 10.1016/0164-1212(94)00111-Y (cited on page 165).

[Tra13] V. Tran Ngoc. "Multi-Database Backends for Ecore Libraries". Master Thesis. Aachen, Germany: RWTH Aachen University, 2013 (cited on pages 168, 190).

[UG96] M. Uschold and M. Gruninger. “Ontologies: Principles, Methods and Applications”. In: *The Knowledge Engineering Review* 11.02 (1996), p. 93. ISSN: 0269-8889. DOI: 10.1017/S0269888900007797 (cited on pages 48, 68).

[Ujh+15] Z. Ujhelyi et al. “EMF-IncQuery: An integrated development environment for live model queries”. In: *Science of Computer Programming* 98 (2015), pp. 80–99. ISSN: 01676423. DOI: 10.1016/j.scico.2014.01.004 (cited on pages 69, 168).

[Vác97] Václav Rajlich. “A Model for Change Propagation Based on Graph Rewriting”. In: *Proceedings of the International Conference on Software Maintenance (ICSM’97)*. 1997, pp. 84–91 (cited on page 108).

[van+03] van der Straeten, Ragnhild et al. “Using Description Logic to Maintain Consistency between UML Models”. In: *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications*. Ed. by J. Stevens, J. Whittle, and G. Booch. Vol. 2863. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 326–340. ISBN: 978-3-540-20243-1. DOI: 10.1007/978-3-540-45221-8_28 (cited on pages 42, 46).

[van+07] van der Aalst, W.M.P. et al. “Business process mining: An industrial application”. In: *Information Systems* 32.5 (2007), pp. 713–732. ISSN: 03064379. DOI: 10.1016/j.is.2006.05.003 (cited on page 73).

[Via16] M. Vianden. *Systematic Metric Systems Engineering: Reference Architecture and Process Model*. 1., Aufl. Aachener Informatik Berichte Software Engineering. Herzogenrath: Shaker, 2016 (cited on page 172).

[Vog13] L. Vogel. *Eclipse IDE: Java programming, debugging, unit testing, task management and Git version conrol with Eclipse*. Vogella series. Lars Vogel, 2013. ISBN: 3943747042 (cited on pages 13, 18, 28, 29, 37).

[vVW07] A. van Deursen, E. Visser, and J. Warmer. *Model-Driven Software Evolution: A Research Agenda*. Delft, 2007. URL: http://www.st.ewi.tudelft.nl/%5Ctextasciitilde%20arie/papers/modse/modse2007.pdf (visited on 02/05/2014) (cited on page 113).

[Wac07] G. Wachsmuth. “Metamodel Adaptation and Model Co-adaptation”. In: *ECOOP 2007 – Object-Oriented Programming*. Ed. by E. Ernst. Vol. 4609. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 600–624. ISBN: 978-3-540-73588-5. DOI: 10.1007/978-3-540-73589-2_28 (cited on pages 34, 45).

[Wag13] S. Wagner. *Software Product Quality Control*. Heidelberg: Springer, 2013. ISBN: 3642385702. DOI: 10.1007/978-3-642-38571-1 (cited on page 13).

[Wal13] R. J. Walker. “Recent Advances in Recommendation Systems for Software Engineering”. In: *Recent Trends in Applied Artificial Intelligence*. Ed. by M. Ali et al. Vol. 7906. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 372–381. ISBN: 978-3-642-38576-6. DOI: 10.1007/978-3-642-38577-3_38 (cited on pages 2, 20, 117).

[War62] S. Warshall. “A Theorem on Boolean Matrices”. In: *Journal of the ACM* 9.1 (1962), pp. 11–12. ISSN: 00045411. DOI: 10.1145/321105.321107 (cited on pages 79, 88).

[WCR05] J. Wohltorf, R. Cissee, and A. Rieger. “BerlinTainment: An Agent-Based Context-Aware Entertainment Planning System”. In: *IEEE Communications Magazine* 43.6 (2005), pp. 102–109. ISSN: 0163-6804. DOI: 10.1109/MCOM.2005.1452837 (cited on page 148).

[Wed01] L. Wedemeijer. “Defining Metrics for Conceptual Schema Evolution”. In: *Database Schema Evolution and Meta-Modeling*. Ed. by H. Balsters, B. Brock, and S. Conrad. Vol. 2065. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 220–244. ISBN: 978-3-540-42272-3. DOI: 10.1007/3-540-48196-6_13 (cited on page 114).

[Wei84] M. Weiser. “Program Slicing”. In: *Proceedings of the 12th annual ACM SIGUCCS conference on User services*. Ed. by R. W. Lutz. New York, NY: ACM, 1984, pp. 439–449. ISBN: 0-89791-146-6. URL: http://dl.acm.org/citation.cfm?id=800078.802557 (cited on page 90).

[Wes14] B. Westfechtel. “Merging of EMF models: Formal foundations”. In: *Software & Systems Modeling* 13.2 (2014), pp. 757–788. ISSN: 1619-1366. DOI: 10.1007/s10270-012-0279-3 (cited on page 43).

[WFK11] K. Wieloch, A. Filipowska, and M. Kaczmarek. “Autocompletion for Business Process Modelling”. In: *Business Information Systems Workshops*. Ed. by W. van der Aalst et al. Vol. 97. Lecture Notes in Business Information Processing. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 30–40. ISBN: 978-3-642-25369-0. DOI: 10.1007/978-3-642-25370-6_4. URL: http://goo.gl/dk8dnp (cited on page 147).

[WHK07] S. Wenzel, H. Hutter, and U. Kelter. “Tracing Model Elements”. In: *IEEE International Conference on Software Maintenance (ICSM 2007)*. 2007, pp. 104–113. DOI: 10.1109/ICSM.2007.4362623 (cited on pages 113, 114).

[Wil06] P. Willett. “The Porter stemming algorithm: then and now”. In: *Program: electronic library and information systems* 40.3 (2006), pp. 219–223. ISSN: 0033-0337. DOI: 10.1108/00330330610681295 (cited on pages 19, 57, 93, 126).

[Wil75] H. J. Will. “Model Management Systems”. In: *Information Systems and Organization Structure* (1975), pp. 468–482 (cited on page 71).

[WK08] S. Wenzel and U. Kelter. “Analyzing Model Evolution”. In: *Proceedings of of the 30th International Conference on Software Engineering*. Ed. by W. Schäfer, M. B. Dwyer, and V. Gruhn. ACM, 2008, pp. 831–834. ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368214 (cited on page 94).

[WKB09] M. Weimer, A. Karatzoglou, and M. Bruch. “Maximum Margin Matrix Factorization for Code Recommendation”. In: *RecSys ’09 Proceedings of the third ACM conference on Recommender systems*. Ed. by L. Bergman and A. Tuzhilin. ACM, 2009, pp. 309–312. ISBN: 978-1-60558-435-5. DOI: 10.1145/1639714.1639775 (cited on pages 9, 48, 117, 144).

[Woh+04] J. Wohltorf et al. “BerlinTainment - An Agent-Based Serviceware Framework for Context-Aware Services”. In: *1st International Symposium on Wireless Communication Systems, 2004*. 2004, pp. 245–249. DOI: 10.1109/ISWCS.2004.1407246 (cited on page 148).

[Woh+12a] C. Wohlin et al. “Empirical Strategies”. In: *Experimentation in Software Engineering*. Ed. by C. Wohlin et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 9–36. ISBN: 978-3-642-29043-5. DOI: 10.1007/978-3-642-29044-2_2 (cited on pages 165, 175).

[Woh+12b] C. Wohlin et al. “Introduction”. In: *Experimentation in Software Engineering*. Ed. by C. Wohlin et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 3–8. ISBN: 978-3-642-29043-5. DOI: 10.1007/978-3-642-29044-2_1 (cited on page 165).

[Woh+12c] C. Wohlin et al. “Systematic Literature Reviews”. In: *Experimentation in Software Engineering*. Ed. by C. Wohlin et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 45–54. ISBN: 978-3-642-29043-5. DOI: 10.1007/978-3-642-29044-2_4 (cited on page 170).

[WS06] J. White and D. C. Schmidt. “Intelligence Frameworks for Assisting Modelers in Combinatorically Challenging Domains”. 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)*. Portland, Oregon USA: ACM Press, 2006, pp. 90–97 (cited on pages 71, 145).

[Wüs14] J. Wüst. *SDMetrics: The Software Design Metrics tool for the UML*. 2014. URL: http://www.sdmetrics.com/ (visited on 04/01/2014) (cited on pages 104, 106).

[WW90] Y. Wand and R. Weber. “An Ontological Model of an Information System”. In: *IEEE Transactions on Software Engineering* 16.11 (1990), pp. 1282–1292. ISSN: 00985589. DOI: 10.1109/32.60316 (cited on page 90).

[YGZ13] K.-H. Yoo, U. Gretzel, and M. Zanker. *Persuasive Recommender Systems: Conceptual Background and Implications*. SpringerBriefs in electrical and computer engineering. New York, NY: Springer, 2013. ISBN: 978-1461447016. DOI: 10.1007/978-1-4614-4702-3 (cited on pages 112, 193, 197).

[Zar+14] Z. Zarwin et al. “Natural Modelling”. In: *The Journal of Object Technology* 13.3 (2014), 4:1. ISSN: 1660-1769. DOI: 10.5381/jot.2014.13.3.a4 (cited on page 17).

[Zic91] R. Zicari. “A Framework for Schema Updates In An Object-Oriented Database System”. In: *Seventh International Conference on Data Engineering*. 1991, pp. 2–13. DOI: 10.1109/ICDE.1991.131447 (cited on page 45).

List of Tables

List of Figures

List of Pseudocodes

Acronyms and Symbols

Only **acronyms** are set in **boldface**.

API Application Programming Interface
AUTOSAR AUTomotive Open System ARchitecture
$\Join$ elements "between" submodels (equation (3.54) (p. 83))
BPMN Business Process Modeling Notation
CDG Class Dependency Graph
CMOF Complete MOF
c context (equation (3.78) (p. 124))
$\mathcal{C}$ universe of contexts
cl cross-link as a tuple of sets (equation (3.54) (p. 83))
DG dependency graph (pseudocode 3.2 (p. 77))
DSL Domain Specific Language
DTD Document Type Definition
Ecore EMF core metamodel
$\mathfrak{Z}^x$ edit sequence (equation (3.82) (p. 124))
ε_x element if denoted, with type x (e.g. p. 36, 55, 83, and 84
E_x elements of type x in Ecore (table 3.1 (p. 34))
E_X elements of type X in knowledge library (table 3.5 (p. 55))
EMF Eclipse Modeling Framework
EMOF Essential MOF
EMG enhanced models graph (equation (3.19) (p. 55))
EMG_E enhanced models graph edges (equation (3.19) (p. 55))
EMG_V enhanced model graph vertices (equation (3.19) (p. 55))
φ find operation (in m) (equation (3.11) (p. 38))
GUI Graphical User Interface
IDE Integrated Development Environment
I_{IDs} index identifiers set (in KL) (equation (3.22) (p. 59))
I index operation (in KL) (equation (3.24) (p. 59))
Ψ index type operation (for KL) (equation (3.39) (p. 62))
KL knowledge library (equation (3.21) (p. 56))
KL_E knowledge library edges (equation (3.21) (p. 56))
KL_I knowledge library indexes (equation (3.21) (p. 56))
$\mathcal{KL}$ universe of knowledge libraries (equation (3.21))
KL_V knowledge library vertices (equation (3.21) (p. 56))
MDE Model-Driven Engineering
MDG Model Dependency Graph
m model as a tuple of sets (equation (3.1) (p. 36))
$\mathcal{MRC}$ universe of model recommendation candidates tuples (equation (3.88) (p. 127))
MP recommendation operation (for KL) (equation (3.76) (p. 120))
ϱ model recommendation operations (equation (3.71) (p. 119))
P model recommendations operations (equation (3.71) (p. 119))
$\mathcal{MR}$ universe of model recommendation tuples (equation (3.110) (p. 133))
$\mathcal{M}$ universe of models (equation (3.1) (p. 36))
MOF Meta Object Facility
OCL Object Constraint Language
OMG Object Management Group
π operation of model operations (equation (3.3) (p. 38))
$\mathfrak{Z}$ operation-based model (equation (3.16) (p. 39))

$\mathcal{M}^{\mathfrak{Z}}$ universe of operation-based models ($\mathfrak{Z}$) (equation (3.16) (p. 39))
OWL Web Ontology Language
QG quality gate (figure 3.27 (p. 103))
Φ query operation (for $\mathbb{KL}$) (equation (3.40) (p. 62))
RCP Rich Client Platform (Eclipse)
RSSE Recommender System for Software Engineering
ρ_x relationships of x in Ecore (table 3.2 (p. 36))
$\mathfrak{R}$ model constraint rules (equation (3.1) (p. 36))
SDK Software Development Kit
σ sequence of model operations (equation (3.15) (p. 39))
S snapshot of a model (figure 3.27 (p. 103))
SPEM Software & Systems Process Engineering Metamodel
s submodel as a tuple of sets (equation (3.47) (p. 75))
$\subseteq$ submodel relation (equation (3.47) (p. 75))
$\rightsquigarrow$ m or `Model` to $\mathfrak{Z}$ transformation (subsection 3.1.4 (p. 40))
τ symbol for trigger method (equation (3.79) (p. 124))
Γ type operation (in m) (equation (3.10) (p. 38))
γ type of a given element (equation (3.10) (p. 38))
UML Unified Modeling Language
UMM Unified Modeling Methodology
URI Uniform Resource Identifier
UI User Interface
UUID Universally Unique Identifier
XMI XML Metadata Interchange
XML Extensible Markup Language

Glossary

basic operations (Π_b) are means of elementary editing for models. They comprise create (π_c), property set ($\pi_{p^{set}}$), relationship assign ($\pi_{r^{ass}}$), and delete (π_d). (equation (3.4) (p. 38))

candidate is an outcome of model recommendation generation that is fed to the ranking and kept as an element of the tuples denoted mrc. (equation (3.89) (p. 127))

`Category` is a concept for arranging `LibraryElements` in a knowledge library. (figure 3.6 (p. 51))

clustering (graph) is an algorithm producing submodel candidates.

concept reuse is the counterpart to software reuse concerning conceptual artifacts, which are project independent. (cf. subsection 2.3.3 (p. 20))

`Connector` is a concept for relating `LibraryElements` in a knowledge library. (figure 3.6 (p. 51))

context is the static environmental information. (page 124 and equation (3.78) (p. 124))

cornflower blue is a popular color in LaTeX, and is sometimes the color of the "better tie at home".

cross-link is a syntactical element of a `Connector` and can restore relationships and other elements that were present before the adjacent `Models` were separated. (subsection 3.3.3 (p. 82))

danube blue is a popular color used by the IBM Rational Software Architect.

dependency graph (DG) is a graph derived from a model either in simple or regular form for clustering. (pseudocode 3.2 (p. 77))

editing sequence describes the editing process subdivided into create, delete, select, and all in operation-based format of the currently edited model, and is part of the context in EDITING as Ξ^x. (equation (3.82) (p. 124))

end of the day "... and here it is, the actual end of the day" [CHM10, End of LFA Review].

evolution is the process of modification of successive generations. (cf. subsection 3.4.2 (p. 95))

evolution graph provides a structure for evolving models as sequences of snapshots. (subsection 3.4.2 (p. 95))

evolution stage automaton structures and builds the foundation for changing stages. (figure 3.25 (p. 100))

evolution step summarizes editing operations so that a change in stages occurs. (subsection 3.4.2 (p. 95))

extent is subdivided into granularity and relatedness for model recommendations. (table 3.11 (p. 121))

`decent` describes a stage that is provisional. (equation (3.68) (p. 100))

`fine` describes a stage that is stable. (equation (3.69) (p. 100))

granularity is one distinction among the extent of model recommendations when parts of models of a `Model` from a knowledge library are leveraged. (table 3.11 (p. 121))

`Group` is a concept for arranging `LibraryElements` in a knowledge library. The `LibraryElements`

might have been used together before, but this is not necessarily the case. (figure 3.6 (p. 51))

Hermes is an Olympian god in Greek religion and mythology, considered a quick and cunning god moving freely between the worlds of the mortal and the divine, e.g., as an emissary and messenger of the gods. He has also been viewed as the protector and patron of, e.g., invention and trade, roads, boundaries, travelers, and thieves. In some myths, he is a trickster and outwits other gods for his own satisfaction or for the sake of humankind.

HERMES is the research prototype implementing operation-based model recommender systems (chapter 4 (p. 151) and section 4.6 (p. 161)). The name is an acronym summarizing the project goals: "Harvest, Evolve, and Reuse Models Easily and Seamlessly".

Indexer is a concept for processing data elements in knowledge libraries for supporting Querys.

Junit also known as Iunit, was a minor ancient Egyptian goddess; unrelated to JUnit a testing framework for Java.

knowledge library is short for an enhanced model graph library. It comprises vertices (KL_V), edges (KL_E), and indexes (KL_I). The vertices are LibraryElements, Models, MetaInformation, and so forth. (table 3.4 (p. 50))

library is a collection of indexed datasets. (cf. table 3.4 (p. 50))

LibraryElement is a concept representing the actual information persisted in a knowledge library. (figure 3.6 (p. 51))

Example is a concept explaining a Model exemplary. (figure 3.6 (p. 51))

marker is part of the harvesting framework for finding known parts ($\mathcal{K}$).

MetaInformation is a concept in a knowledge library providing additional information regarding quality. (figure 3.6 (p. 51))

metamodel is the model of a model, or an abstraction of the modeled concepts. (cf. subsection 2.3.1 (p. 16) and subsection 2.5.1 (p. 28))

model is an abstraction of real-world elements. In our case, the term is often a synonym for UML class diagrams. (cf. subsection 2.3.1 (p. 16))

Model is a subconcept of LibraryElement representing actual Ecore models. (figure 3.6 (p. 51))

model elements are the basic building blocks of models, e.g., classes and attributes; by and large, they comprise MOF or Ecore concepts. (table 3.1 (p. 34))

model operations are our means of representing the editing sequences of models. They comprise basic and supporting operations. (equation (3.3) (p. 38))

model recommendation is an information item of estimated value for modeling. (see page 119)

model relationships are the basic connectors of models, e.g., references; by and large, they comprise MOF or Ecore relationships. (table 3.2 (p. 36))

model reuse is a process implementing reuse for model. (cf. subsection 2.3.3 (p. 20))

model, operation-based see operation-based model

ontology is a formal, explicit specification of a shared conceptualization [Gru93]. (subsection 3.2.4 (p. 65))

operation-based model ($\mathfrak{Z}$) is a representation of a model in a sequenced manner. It comprises model operations. (equation (3.1) (p. 36))

operations – see model operations or recommendation operation

plug-in name for an OSGi bundle in Eclipse. (cf. subsection 2.5.1 (p. 28))

proactive describes model recommendation presentation when deemed appropriate [Rob+14, Glossary]. (subsection 3.5.3 (p. 122))

proactive quality guidance means to qualitatively guide model editing.

production is to be considered in the context of model recommendations and comprises the steps undertaken to gain model recommendations. ($\mathfrak{MP}$ equation (3.77) (p. 120))

production sequence comprises operations undertaken during model recommendation production. (equation (3.76) (p. 120))

`purpose` is an attribute of `MetaInformation` meant to provide a lightweight specification. (figure 3.6 (p. 51))

quality characteristic are leaves in a quality model. (figure 3.26 (p. 102))

quality gate (figure 3.27 (p. 103)).

quality measure are means for assessing model properties.

`Query` is a concept for leveraging data prepared by `Indexer` for retrieval.

reactive describes model recommendation presentation when requested [Rob+14, Glossary]. (subsection 3.5.3 (p. 122))

recommendation operation helps subdivide model recommendation production into analyze, generate, rank, and filter operations. (equation (3.71) (p. 119))

recommender system is an information filtering system for "big data" tasks attempting to predict preferences or actions based on known preferences or actions from historical/similar data. (cf. subsection 2.3.2 (p. 19))

relatedness is one distinction among the extent of model recommendations when non-syntactical information from a knowledge library is leveraged. (table 3.11 (p. 121))

repository is usually a storage location. (cf. subsection 2.3.1 (p. 16))

reuse is commonly used as a sloppy abbreviation for reutilization. More precisely, it is a multistep process leading to reutilization. (cf. subsection 2.3.3 (p. 20))

reutilization is the last step of model reuse, considering it a process and putting models in a new environment. In general, the term reuse is used as a synonym for reutilization, which is more precise.

saver is part of the harvesting framework for persistence.

schema are best practices for model recommendation production and generation. (subsection 3.5.7 (p. 138))

sensitivity comprises one type of model recommendation production operation concentrating on querying a knowledge library. (equations (3.93) to (3.98) (p. 128))

sequence (evolution) is a successive series of model snapshots. (subsection 3.4.2 (p. 95))

sequence can be a summarizing concept for model model operations as a sequence of operations, model evolution as an sequence, model recommendation production as a production sequence, or model editing sequence (Ξ^x).

sequence of operations (σ) is a convenient concept to summarize more operations that are meant to be read, ordered from right to left. (equation (3.15) (p. 39))

snapshot (evolution) is a point in time over the evolution of a model. (subsection 3.4.2 (p. 95))

software reuse is the counterpart to concept reuse concerning software artifacts, which are project specific. (cf. subsection 2.3.3 (p. 20))

splitter is part of the harvesting framework for creating submodel candidates.

stage (evolution) describes the evolution status of a model with respect to reusability and quality.

submodel is a subset of a model that is syntactically complete and sound. (subsection 3.3.2 (p. 76))

submodel candidate is a potential submodel produced by clustering to be manually altered or stored as a valid model represented by a `Model` in a knowledge library.

supporting operations (Π_s) are means of helping elementary editing for models. They comprise type identification (Γ), find in models (φ), update (π_u), and revert (π_{-1}). (equation (3.9) (p. 38))

`TemplateInformation` is a concept for generalizing `Models`. (figure 3.6 (p. 51))

`vague` describes a stage that is sketchy. (equation (3.67) (p. 100))

Index

F

G

H

I

K

Q

R

S

T

U

V

X

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

`http://aib.informatik.rwth-aachen.de/`

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: biblio@informatik.rwth-aachen.de

2015-01 *	Fachgruppe Informatik: Annual Report 2015
2015-02	Dominik Franke: Testing Life Cycle-related Properties of Mobile Applications
2015-05	Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder: Inferring Lower Bounds for Runtime Complexity
2015-06	Thomas Ströder and Wolfgang Thomas (Editors): Proceedings of the Young Researchers' Conference "Frontiers of Formal Methods"
2015-07	Hilal Diab: Experimental Validation and Mathematical Analysis of Cooperative Vehicles in a Platoon
2015-08	Mathias Pelka, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 1st KuVS Expert Talk on Localization
2015-09	Xin Chen: Reachability Analysis of Non-Linear Hybrid Systems Using Taylor Models
2015-11	Stefan Wüller, Marián Kühnel, and Ulrike Meyer: Information Hiding in the Public RSA Modulus
2015-12	Christoph Matheja, Christina Jansen, and Thomas Noll: Tree-like Grammars and Separation Logic
2015-13	Andreas Polzer: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen
2015-14	Niloofar Safiran and Uwe Naumann: Symbolic vs. Algorithmic Differentiation of GSL Integration Routines
2016-01 *	Fachgruppe Informatik: Annual Report 2016
2016-02	Ibtissem Ben Makhlouf: Comparative Evaluation and Improvement of Computational Approaches to Reachability Analysis of Linear Hybrid Systems
2016-03	Florian Frohn, Matthias Naaf, Jera Hensel, Marc Brockschmidt, and Jürgen Giesl: Lower Runtime Bounds for Integer Programs
2016-04	Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder: Proving Termination of Programs with Bitvector Arithmetic by Symbolic Execution
2016-05	Mathias Pelka, Grigori Goronzy, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 2nd KuVS Expert Talk on Localization
2016-06	Martin Henze, René Hummen, Roman Matzutt, Klaus Wehrle: The SensorCloud Protocol: Securely Outsourcing Sensor Data to the Cloud

2016-07	Sebastian Biallas : Verification of Programmable Logic Controller Code using Model Checking and Static Analysis
2016-08	Klaus Leppkes, Johannes Lotz, and Uwe Naumann: Derivative Code by Overloading in C++ (dco/c++): Introduction and Summary of Features
2016-09	Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, Peter Schneider-Kamp, and Cornelius Aschermann: Automatically Proving Termination and Memory Safety for Programs with Pointer Arithmetic
2016-10	Stefan Wüller, Ulrike Meyer, and Susanne Wetzel: Towards Privacy-Preserving Multi-Party Bartering
2017-01 *	Fachgruppe Informatik: Annual Report 2017
2017-02	Florian Frohn and Jürgen Giesl: Analyzing Runtime Complexity via Innermost Runtime Complexity
2017-04	Florian Frohn and Jürgen Giesl: Complexity Analysis for Java with AProVE
2017-05	Matthias Naaf, Florian Frohn, Marc Brockschmidt, Carsten Fuhs, and Jürgen Giesl: Complexity Analysis for Term Rewriting by Integer Transition Systems
2017-06	Oliver Kautz, Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe: CD2Alloy: A Translation of Class Diagrams to Alloy
2017-07	Klaus Leppkes, Johannes Lotz, Uwe Naumann, and Jacques du Toit: Meta Adjoint Programming in C++
2017-08	Thomas Gerlitz: Incremental Integration and Static Analysis of Model-Based Automotive Software Artifacts
2017-09	Muhammad Hamad Alizai, Jan Beutel, Jó Ágila Bitsch, Olaf Landsiedel, Luca Mottola, Przemyslaw Pawelczak, Klaus Wehrle, and Kasim Sinan Yildirim: Proc. IDEA League Doctoral School on Transiently Powered Computing
2018-01 *	Fachgruppe Informatik: Annual Report 2018
2018-02	Jens Deussen, Viktor Mosenkis, and Uwe Naumann: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen
2018-03	Igor Kalkov: A Real-time Capable, Open-Source-based Platform for Off-the-Shelf Embedded Devices

* These reports are only available as a printed version.
Please contact biblio@informatik.rwth-aachen.de to obtain copies.