Staged model evolution and proactive quality guidance for model libraries

Andreas Ganser¹ · Horst Lichter¹ · Alexander Roth² · Bernhard Rumpe²

Published online: 25 November 2015
© Springer Science+Business Media New York 2015

Abstract A variety of modeling approaches, including model-driven development, consider model reuse as one of their cornerstones, but lack support for model reuse. This may be due to the available model repositories that barely exceed support for enhanced versioning or collaborative work and disregard model evolution. We believe that current model evolution approaches do not consider reuse sufficiently and that model repositories for reuse purposes should act as model libraries. This requires new functionality, because models for reuse need to achieve and maintain high quality. Moreover, quality assessment and assurance, which are tasks often considered tedious, need to be as simple as putting away or maintaining artifacts for reuse. In this study, we propose an approach for model evolution in UML model libraries that differs from general model evolution, since it is aimless and triggered by new external requirements. Our approach is founded on graphs that are partitioned into three stages with respect to the level of reusability. Each stage is defined by quality characteristics that are manifestations of a quality model consisting of four essential quality dimensions: syntactic, semantic, pragmatic, and emotional. In order to achieve the next level of reusability, i.e., change the stage of a model, a quality gate needs to be passed. This can be supported by a proactive approach that guides the modeler through the enhancement process and offers additional recommendations based on the level of reusability. Since guidance cannot be fully automated, we implement a review mechanism founded on the idea of the six thinking hats to help maintain focus on the main
aspects of a review. Finally, our approach is enhanced to support the evolution of generations, i.e., a group of several model snapshots, to ease reusability.

**Keywords** UML · Model · Evolution · Quality · Libraries

1 Introduction

Modeling is a traditional discipline in computer science. Computer scientists have been creating models for decades and have seen models evolve into a plethora of different forms. Interestingly, the general modeling theory was not developed by a computer scientist, but rather by the philosopher Herbert Stachowiak in the 1970s (Stachowiak 1973). His work impacted many areas of computer science, such as databases, and resulted in different research areas, including conceptual modeling (Embley and Thalheim 2011) and generic model management (Melnik 2004). The latter transformed Stachowiak’s abstract theories into an applicable approach offering concepts and algorithms, such as diff, merge, similarity, and match operations. Consequently, pure model operations were investigated with a broader perspective (Embley and Thalheim 2011).

A similar development occurred in object-oriented modeling, which established unified modeling language (UML) as a suitable modeling language. Today, the success of UML is often accredited to two things: (1) UML is an effective language, because, “for larger and distributed projects, UML modeling is believed to contribute to a shared understanding of the system and more effective communication” (Chaudron et al. 2012); (2) UML is considered the de facto standard for modeling and, as a result, many tools have been developed involving UML, including code generators. These tools bolster approaches, such as rapid prototyping, agile modeling (Rumpe 2012), and model-driven development (MDD), and allow modelers to deal with complexity at an appropriate level of abstraction.

Consequently, UML models are widely used and can be regarded as project assets that should be reused. It is also believed that model reuse can decrease development time while increasing software quality (Mens et al. 1998; Lange and Chaudron 2005) by leveraging best practices and experiences. The main question involves how to store models in a way that enables modelers to improve their quality over time. Certainly, model reuse requires an infrastructure enabling models to persist in a library or knowledge base. Furthermore, reuse needs a means of controlling model evolution and quality in the long term. Unfortunately, quality is subjective, often relative to requirements, and sometimes hard to measure (Moody 2005). Moreover, all-at-once quality assessments result in endless quality reports that are difficult to work through. One alternative is edit-time quality assessment and guidance for assuring a certain level of quality in model libraries, a method we call proactive quality guidance. To the best of our knowledge, this is the first approach of its kind.

We reviewed recent research (Sect. 8) and found that model evolution is often considered a goal that would be self-defeating in model libraries. We adapted the meaning of model evolution to fit model libraries and developed an approach (Sect. 3.1) that explains how models should evolve within model libraries. This enabled us to discuss model evolution in model libraries on a more formal level by means of model evolution graphs (Sect. 3.2). Moreover, we introduced a structure called evolution stages (Sect. 3.3) to define a formal basis that would help structure their relationships in an automaton...
We then introduced a quality model for the models (Sect. 4.1), which is linked to the automaton in order to define gates (Sect. 4.2) between stages. We established these gates by introducing quality measurement instruments (Sect. 4.3) and proactive quality assessments (Sect. 5) to help automate transitions in the automaton. Finally, we extended our approach to a range broader than individual models. Therefore, we studied how model evolution works in model libraries (Sect. 7.1) and allows models to be grouped as generations (Sect. 7.2). Additionally, we assessed what impact analysis (Sect. 7.3) means in model libraries.

2 The HERMES project

The model evolution approach presented here is built for model libraries and is part of the Harvest, Evolve, and Reuse Models Easily and Seamlessly (HERMES) project (Ganser 2014c, b). It comprises four parts to harvest, evolve, and reuse models easily and seamlessly, as shown in Fig. 1. We briefly explain each part in order to introduce the environment and provide the basis for the requirements that the evolution approach will need to meet. These parts sometimes need to be illustrated in combination with the vision that initiated the project. This vision is sketched in Fig. 2, which shows the mechanism of code completion for graphical and textual modeling environments.

Figure 2 shows a class diagram editor with a search box (cf. Ganser 2014a). A user inputs letters, and the system returns results presented as preview items in a drop-down box that the user can step through. As soon as the user chooses an item, it is applied to the canvas along with other manually created elements, such as classes, attributes, and associations. This completion mechanism works similar to other code-completion mechanisms found in state-of-the-art programming environments.

At first glance, the mechanisms underlying code completion and model completion seem very similar; however, the two systems differ significantly. The code-completion mechanism relies on the grammar of the programming language, available source code, and libraries, while model completion proves appealing only if a model library is used that provides more than syntactical support. Such a library needs to contain best practices, patterns, examples, or partial models (Dyck et al. 2014b). As a consequence, we have developed a model library that is capable of storing all of this information and more in an enhanced knowledge graph (Ganser and Lichter 2013).

The second difference between code completion and model libraries concerns the fact that there is almost no research on how to determine the best model for a given
environment, which we call context. The most promising approach involves adapting ideas from recommender systems (Jannach et al. 2011; Ricci et al. 2011), given that model libraries tend to contain much information which is not immediately apparent. We developed an environment for experimenting with model recommenders (Dyck et al. 2014a; Ganser 2014c). These experiments may address how to apply the selection presented in Fig. 2 to different editors, how to use different data sources, or how to gain enough information from the context to enable the recommender algorithm to produce “good” reactive and proactive recommendations.

A third difference involves the means by which models are found for the model library and, subsequently, how the model library is filled. Enhanced code completion can use data mining techniques and analyze existing source-code repositories in order to fill databases and optimize code recommenders (Weimer et al. 2009; Eclipse Foundation 2014a). This enables the mined data to be leveraged to reorder the available completions based on different possibilities. Contrary to this, there are no data mining approaches available for models. Nevertheless, tool support can help users and we are working to develop a harvesting mechanism that adapts the methods inherent to data mining.

Unfortunately, there is no guarantee that a model freshly stored in the model library will suit the needs of a modeler. Consequently, an approach is required that aims for model evolution in model libraries, leading to high-quality models by tracking models over time (Roth et al. 2013b) and guiding changes by means of real-time quality feedback (Roth et al. 2013a) with simple quality statements. Eventually, this approach should foster model reuse by model recommenders.

In a summarizing figure, which alters Fig. 1, our contribution is framed by our working context in Fig. 11.

3 Model evolution approach

General model evolution differs from model evolution in model libraries, because it is aimless and triggered by new external requirements (Roth et al. 2013b). This means that unlike general model evolution, evolution in model libraries has reuse as a goal and avoids changing requirements regarding specific models. As a consequence, models in model libraries do not strive for perfection in every possible deployment scenario, but rather endeavor for an adaptable, comprehensible, and appropriate solution that fits 80% of the
requirements. A sound foundation for model evolution in model libraries is, however, required to enable tool support that helps modelers design reusable models.

Before evolution in model libraries can begin, models need to be initialized. This means that a model under examination must contain parts that are generic or generally reusable, enabling them to be added to a model library. The steps involved include extracting parts of a given model, preparing them for reuse, and storing the resulting model in a model library. The last step requires modelers to annotate the extracted model with a descriptive specification called model purpose. This is supposed to describe the primary intention of the model and reflect its general purpose in a few words. By attaching this complementary description and adding the underlying model to the model library, the evolution of this particular model in the model library begins.

3.1 Model evolution basics

A foundation for model evolution in model libraries can be built upon Lehman’s description of software evolution as “a progressive and beneficial process of changes in evolving attributes of an entity” (Lehman 1980). For models, progressive changes are add, delete, rename, and retype operations, with the semantics defined by Herrmannsdoerfer et al. (Herrmannsdörfer and Ratiu 2010). A similar understanding is presented by Keienburg et al., who defined evolution as “proceeding an ordered list of model change primitives on an existing component model [to] finally create a new model version” (Keienburg and Rausch 2001).

Therefore, model evolution in model libraries is based on different model versions that we refer to as model snapshots. Changes to a model produce new model snapshots, which consist of the resulting model, including the changes. Two consecutive model snapshots of the same model, i.e., a model snapshot and the resulting model snapshot after applying a set of actions, form an evolution step. Every set of model snapshots contained in a list of consecutive evolution steps represents model evolution for a particular point in time. Hence, the empty set represents no evolution, and each non-empty set of model snapshots defines model evolution up to the latest model snapshot included.

An example of an evolving class diagram representing a simplified Airport model is shown in Fig. 3. In “Snapshot 1”, the evolution of the model begins, then changes due to rename, retype, delete, or add operations until it reaches “Snapshot 3”. Further changes may lead to more snapshots and further model evolution.

The model evolution described above is not solely based on adaptations and is different from maintenance. Still, adaption and maintenance are major influences on model
evolution, even if model evolution is understood as undirected, infinite, and unpredictable (Briand et al. 2006; Eick et al. 2001; Madhavji et al. 2006). The primary concerns for maintenance are correcting faults, adapting, improving a model, and preserving integrity between models and source code (Judson et al. 2004). Maintenance is not the exclusive reason for model evolution, given that models can evolve due to structural changes enforced by successive development rather than sustainment. However, maintenance changes models and, thereby, influences model evolution.

Judson et al. defined three types of evolution (Judson et al. 2004). First, adaptive evolution occurs due to changes in requirements. Second, perfective evolution comprises enhancements to model design quality. Finally, corrective evolution summarizes error corrections. Each type of model evolution characterizes reasons for evolution. This differs from maintenance, which describe evolution activities.

3.2 Model evolution graph

Our more theoretical foundation of model evolution in model libraries is similar to version-control graphs, where evolutionary progress is structured and mapped to a graph structure. This simplistic, but sufficiently formal representation serves as the foundation for our tool (Fig. 4).

In our approach, model evolution consists of two essential parts: model snapshots and evolution steps. These are based on a model evolution graph, whose vertices are model snapshots and where edges represent evolution steps. Each vertex is labeled with a unique version id and at least two adjacent edges labeled with a transition id. An example of a model evolution graph is illustrated at the bottom of Fig. 3, which shows that the evolution

Fig. 4  Software prototype of the model evolution stage monitor and the corresponding model
of the simplified Airport model starts in a snapshot labeled with a version id $S1$ and progresses to snapshot $S3$.

Each edge label in the model evolution graph represents a sequence of primitive operations applied to the model. These transition the model to the next model snapshot according to the direction of the edge. In order to comply with Sect. 3.1, these primitive operations are defined as $\tau_{\text{add}}, \tau_{\text{delete}}, \tau_{\text{rename}},$ and $\tau_{\text{retype}}$, and a sequence of primitive operations is $\sigma_i \in \{\tau_{\text{add}}, \tau_{\text{delete}}, \tau_{\text{rename}}, \tau_{\text{retype}}\}^*$, where $i \in \mathbb{N}$, forms the edge labeling.

In all, the model evolution graph is a tuple, $(EG_{model}, s, t)$, where $EG_{model} = (V, E)$ represents a graph with a vertex set $V$, an edge set $E$, and two labeling functions, $s$ and $t$. $s : V \rightarrow S$ is a function for some domain, $S$, labeling each vertex. Each edge is labeled by the function $t : E \rightarrow T$, where $T = \{\tau_{\text{add}}, \tau_{\text{delete}}, \tau_{\text{rename}}, \tau_{\text{retype}}\}^*$. For the remainder of this article, model evolution will be discussed with respect to this underlying model evolution graph. Specifically, it will be regarded as a sequence of vertices of a set of consecutive edges in a model evolution graph.

Currently, model evolution based on model evolution graphs does not directly contribute to model reusability. Hence, team discussion with seven participants was carried out to determine how to merge reusability concerns with the model evolution graph (see Sect. 6). The goal was to establish how to add status information for each model snapshot. Such status information can then be used to partition the model evolution graph into sequences of snapshots containing qualitative information about reusability.

### 3.3 Model evolution stages

Given the results of our user study, we extended each model snapshot to include additional status information regarding modeler concerns, including improvement suggestions and other information for modelers. Hence, model snapshots can be grouped by tracking a concern that is currently relevant to modelers until it is no longer relevant. Since the main focus of model libraries is reusability, we subdivided the concerns into three levels of reusability, namely low, medium, or high. This means that every reusability concern describes the current reusability level of the model and comprises status information relevant to this reusability level.

By rooting qualitative evaluation of reusability concerns on the traffic-light metaphor, cognitive load can be neglected, due to the number of options being small and the semantics associated with traffic-light colors being obvious. Moreover, we grouped consecutive sequences that follow the same goal to achieve the same reusability status and mapped the result to one of the three reusability concerns in order to form stages (vague, decent, and fine). The stages comprise successive model snapshots that build upon one another to achieve subsequent reusability stages.

A vaguely reusable model is undetermined with respect to its reusability and, thus, has a red label, denoting that the model requires further processing. Nevertheless, it is offered for reuse; however, given the uncertainty of its reusability, it should be handled with great care. Different concerns, including specific suffixes/prefixes, technology-dependent elements, adapters for legacy use, or even errors, might have to be resolved. Altogether, the “vague” stage represents the starting point of model evolution and contains models that are generally reusable, but require improvements.

As major issues are resolved, the model progresses toward a sufficiently reusable model and is labeled “yellow.” This model does not contain any specifics or errors and can be reused in most scenarios. Still, certain care should still be taken. For instance, the purpose
of the model may not perfectly match the requirements or design decisions may have to be improved upon. Furthermore, the layout may be unappealing or the qualitative statements might rely on assessments alone rather than actual experience while reusing this model.

Finally, a highly reusable model, i.e., a model in the “fine” stage, is considered almost perfectly reusable and holds a “green” label. The purpose of the model is in line with the model itself, and all known errors have been resolved. However, models will likely not be reusable unless adoptions are made. One reason is template mechanisms, which require a modeler to fill in optional requirements or adoptions required by this particular reuse.

According to the model properties for each stage, the required skill set of a modeler can be estimated. Derived from the Dreyfus model of skill acquisition, we defined the skill stages as novice, competent, and expert modeler stages (Dreyfus and Dreyfus 1980). Therefore, regarding reusing a model, an expert might be able to use all of the models available, but a competent modeler would not be able to use “vague” reusable models, and a novice modeler would not be able to use “vague” or “decent” reusable models. However, the expert user will still be provided stage information, e.g., the “red” sign as a warning.

These three stages form a simple, yet intuitive foundation, but experience from programming application-programming interfaces serves as a reminder that model artifacts may need to be locked and made unavailable at some point. Hence, we enhanced the stages by adding a flag titled deprecated, which labels a model in its current state as retired or no longer recommended for use. Despite this, the model still retains its status to serve in some capacity in the future, e.g., as a counterexample.

### 3.4 Model evolution automaton

In staged model evolution, a “vague” reusable model cannot become highly reusable unless its purpose as a model and the design decisions are validated during the “decent” stage. This entails an underlying order to staged model evolution, which is defined by an automaton.

The staged model evolution automaton, as shown in Fig. 5, is defined as $A_{staged} = (Q, \Sigma, Z, \delta, q_0, F)$, where $Q := \{ VAGUE, DECENT, FINE \}$ is the set of vertices, $\Sigma := Q \times (T \cup \{ id_M \})$, and with $T := \{ s_{add}, s_{delete}, s_{rename}, s_{retype} \}$ as the set of input symbols, $q_0 (q_0 := VAGUE)$ as the starting state, and $F (F := \emptyset)$ as the empty set, since

![Fig. 5 Model evolution stages](https://example.com/fig5.png)

© Springer
evolution is not regarded as being finite. The transition, $\delta$, is defined as $\delta := \delta_v \cup \delta_d \cup \delta_f$, with

$$\delta_v := \{(\text{VAGUE}, \tau_i) \rightarrow \text{VAGUE} \mid \tau_i \in T\}$$

$$\cup \{(\text{VAGUE}, \text{ID}_M) \rightarrow \text{DECENT}\},$$

$$\delta_d := \{(\text{DECENT}, \tau_i) \rightarrow \text{DECENT} \mid \tau_i \in T\}$$

$$\cup \{(\text{DECENT}, \text{ID}_M) \rightarrow \text{VAGUE}\}$$

$$\cup \{(\text{DECENT}, \text{ID}_M) \rightarrow \text{FINE}\}, \text{AND}$$

$$\delta_f := \{(\text{FINE}, \tau_k) \rightarrow \text{FINE} \mid \tau_k \in \{\text{rename}, \text{retype}\}\}$$

$$\cup \{(\text{FINE}, \text{ID}_M) \rightarrow \text{DECENT}\}$$

$$\cup \{(\text{FINE}, \text{ID}_M) \rightarrow \text{VAGUE}\}$$

The initial stage for each model is the “vague” stage. In each stage, certain operations ($\tau$) can be performed until an identity operation ($\text{ID}_M$) moves the model to another stage. This activity is a technicality at this point in the process and will be expanded upon later in the discussion of quality gates. For now, identity operations are a means for snapshots. The impact on the model varies, as the operations available on a model in the “fine” stage are limited, because adding or deleting model elements in this stage could change the model entirely.

Although a formal foundation is given by the model evolution graph, non-determinism is inherent. All outgoing transitions in the “fine” stage can represent the same type of operations. Moreover, identity operations are not a hindrance between stages because it is always possible to change stages. However, additional information can be leveraged to ensure that models only move to the next stage if certain requirements are fulfilled.

### 3.5 Model evolution in action

The automaton underlying staged model evolution describes how models evolve in model libraries based on stages. Due to its non-determinism, staged evolution is controlled by the modeler. A detailed example of this evolution is depicted in Fig. 3, which illustrates how models may evolve. Additionally, it points to the need for a clear separation between stages, as well as guidance.

After the simplified Airport model is assumed to be beneficial for reuse, a modeler adds it to the model library. Specifically, it was extracted from its environment and stored for reuse, as described in Sect. 3. The model starts in the “vague” stage. This is very reasonable because the model contains errors and the attribute name has no type. Thus, before moving any further, this error needs to be fixed.

Following error resolution, the model can progress to the “decent” stage. Additionally, the modeler also decides to rename a class and add four additional classes. The resulting model is “Snapshot 2” in Fig. 3. Although this model is reusable, in general, the design may need to be revised. With respect to this being a model of a simplified Airport, the model contains a Person class, which does not fit the purpose of the model. After deleting this class, the modeler can move the model to the “fine” stage, wherein the model is coherently valid and can be reused by other modelers.

Due to “add” and “delete” operations being available for model alteration, the purpose and validity of the model may change entirely. These actions lower both the reusability level and stage of the model, respectively. In order to help modelers and to enforce accurate traversal of staged model evolution, a clear separation of the stages is beneficial.
4 Quality staged model evolution

According to the staged model evolution automaton, a modeler may non-deterministically change the stage of a model at any time without fulfilling any constraints. In other words, the semantics of the stages do not yet adequately describe the model. In order to address this, we defined a set of measurable properties, wherein each set is defined for one particular stage and holds true for a model within that stage. The “vague” stage has no set of predefined properties determined, given that it is the initial stage.

These sets of properties are subsets of the quality characteristics summarized in a quality model and form quality gates for each state-changing transition in the staged model evolution automaton. Consequently, we extended the staged model evolution approach with quality characteristics based on qualitative concerns for separating stages.

4.1 The quality model for models

Generally, quality models are difficult to define. Specifically, the underlying quality characteristics are difficult to measure due to their highly subjective and relative nature (Moody 2005). Nevertheless, model evolution in model libraries becomes manageable, because only general characteristics, such as model size or complexity, are important.

Our quality model is based on the research of Lindland et al. (1994) and Lange and Chaudron (2005) and comprises four general quality dimensions: syntactic, semantic, pragmatic, and emotional quality. Each can be subdivided as shown in Fig. 6. If a model in a model library conforms to these general quality characteristics, it is considered to be of high quality and reusable.

Syntactic quality can be used for error prevention and error detection. Error correction is, however, unfeasible due to the uncertainty of modeler goals. It is sufficient to restrict

![Model quality dimensions and quality characteristics](image-url)

Fig. 6 Model quality dimensions and quality characteristics
syntactic quality to three characteristics in order to focus on reusability. *Meta-model conformity* focuses on the conformity of the model to the abstract syntax specifying the modeling language, i.e., the model is valid [an eclipse modeling framework (EMF) model conforms to Ecore]. *Transformability* requires the existence of a generator that successfully transforms the examined model into another representation, e.g., source code or commands applicable to graphical modeling environments, such as EMF or graphical modeling framework editors (Dyck et al. 2013). Finally, a model without any syntax errors is *defect-free*, e.g., a well-formed XML document with a tree structure.

With respect to the purpose of the original model, semantic quality is considered to the extent that the model describes the item that it should model, i.e., the correspondence of a model and its domain. Semantic quality is subdivided into three model quality characteristics: *semantic validity, completeness, and confinement*. A model is of high semantic validity if each and every statement described by the model is valid with respect to a domain. An invalid statement can only be contained if removing it results in a higher advantage relative to the drawback of adapting the model without removal of the invalid statement. In this case, the invalid statement is simply removed. In contrast, the drawback associated with not including each statement of the domain in the model must be greater than the advantage of simply neglecting each statement. If this holds for all statements in the domain, the model is complete. Lastly, if a model includes enough valid statements about the domain to describe the intention or to solve a problem in a domain, the model is “fine”.

Another model quality dimension that concerns the degree to which the model is understood by modelers is pragmatic quality. This addresses the correspondence between modeler comprehension and the model itself. Given the focus on model libraries, model comprehension can be reduced to three characteristics. The most intuitive is understandability, which refers to the degree to which modeler interpretation can be derived from the model. Maintainability of the model is given if the modeler understands the model and is able to adapt it to environmental changes. Additionally, the model purpose extraction characteristic is fulfilled if the purpose of the model extracted by the modeler is in line with the purpose of the original model as explained below.

Emotional quality is an unconventional quality dimension. It addresses the correspondence between user interpretation and user emotion. We believe that highly appealing models are more likely to be reused and, thus, have a high emotional quality. In model libraries, this model quality is important due to the individual opinions of the modelers. This rather imprecise quality dimension, we only mention for the sake of completeness, is represented by the round shape in Fig. 6, and approaches for measuring emotional quality are shown in Fig. 4 (cf. “thumb-up button”).

![Staged model and quality gates](image-url)
4.2 Quality gates

The quality model comprises four dimensions that characterize quality concerns for models in model libraries and, thereby, derive their reusability. The staged model evolution automaton aims at structuring and guiding model evolution in model libraries. By assigning a set of model quality characteristics from Fig. 6 to each stage transition in the automaton, quality gates are defined. As a consequence, modeler subjectivity during staged model evolution is reduced and the evolution process guided.

A model may pass a quality gate by fulfilling the requirements associated with that particular gate. Our approach incorporates several quality gates into the staged model evolution automaton, as depicted in Fig. 7, and an overview of the assignments between model quality characteristics and quality gates is illustrated in Fig. 11.

Each model starting in the “vague” stage aims to reach the “fine” stage by fulfilling every model quality characteristic. When a modeler intends to move the model to the “decent” stage, the first quality gate (QG1 in Fig. 7) needs to be passed. This requires meta-model conformity, transformability, semantic validity, confinement, and status as defect-free. Passing QG1 validates that (1) no prefixes or technological dependencies are included, (2) legacy adapters are removed, and (3) the model has no errors. This is indicated by a tailing “d” in Fig. 6. Please note that semantic completeness is not required, because it is targeted in the “decent” stage.

In the “decent” stage, staged model evolution can proceed in two possible directions. The model can be moved to the “fine” stage, if the purpose of the model is in line with the original model purpose and if it provides a satisfying solution for modelers. In this case, the subsequent level of reusability is “green,” i.e., the model library may suggest this model to modelers for reuse. To reach this stage, the quality gate (QG3) must be passed, which requires that all quality characteristics be fulfilled (all quality characteristics with a tailing “f” in Fig. 6). However, emotional quality is not considered a requirement related to the number of “likes”. Instead, these criteria indicate that modelers agreed on the reuse of the model and its quality. On the other hand, if the model contains errors, needs restructuring, or is unsuitable for reuse, it can be moved back to the “decent” stage by passing quality gate (QG2). This requires violation of at least one of the requirements for QG1.

When a model reaches the “fine” stage, it becomes reusable with respect to the quality model described in Sect. 4.1. Different internal and external influences, including operations other than renaming or retyping, may, however, change the model reusability level. Depending on the type of adaptions, the model needs to be moved to the “decent” or “vague” stage. Model restructuring, which leads to model decomposition or even new models, requires the model to be moved to the “vague” stage. Bugs that are either difficult to fix or will not be fixed also push the model into the “vague” stage. For any other adaption, the model needs to be moved to the “decent” stage. Both cases require the model to pass quality gates (QG5 or QG4, described in Fig. 7). Each of them requires at least one violation of the requirements for QG3. Please note that the “fine” stage is non-deterministic, because the modeler needs to determine the impact of the adaptions and move the model to the corresponding stage.

It is important to note that the evolution process must never be fully automated, because the modeler may object to a passed quality gate. This is why a stage might qualify as pending, as shown in the screenshot in Fig. 4. Hence, the final determination of model stage status requires confirmation by a modeler.
In summary, our quality gates enhance staged model evolution by providing clear separation between each stage. In order to provide tool support and make stage evolution practicable for modelers, each model quality characteristic needs to be measured. Currently, automatic and manual quality measurement approaches are offered to measure model quality characteristics. Please note that appeal is not measured, but is tracked using a counter that counts the number of model reuses.

4.3 Quality measurement instruments

Each model quality dimension, especially the underlying characteristics, has to be measured in order to enable qualitative statements about model quality in model libraries. Evaluating model quality characteristics reveals that only some are automatically determinable. Certainly, syntactic errors can be found by parsers. In contrast, completeness is highly subjective and hard to measure. Consequently, distinctions are made regarding model quality measurement instruments across three categories: strong, medium, and weak characteristics. Such classification allows a mapping from model qualities (cf. Fig. 6) to quality measurement instruments, where each characteristic is used to obtain feedback with respect to the name of the characteristic.

The first type of characteristic (strong) is the strictest and can be precisely measured using model metrics, such as SDMetrics (Wüst 2014) and (Genero et al. 2002). Each model metric is formulated with respect to models and provides clear feedback, including the cause metric failure and a suggested solution. For instance, a model including a class

![Fig. 8 Simple review foundation and realization. a Five simple review hats. b Simple review on Fig. 4](image)
without a name constitutes a measurable model metric. Existing tools allow measurement of such “strong” characteristics, e.g., EMF validator and EMF generator (Steinberg et al. 2008). With respect to the quality model shown in Fig. 6, “strong” characteristics include defect-free status, meta-model conformity, and meta-model transformability.

Our “medium” characteristics are based on Fowler’s idea of code smell (Fowler 1999). A smell is something that seems to be wrong, can be measured in some way, and should be improved. For instance, a model with hundreds of classes is harder to understand than one with only a few. Again, model metrics having a clear threshold can be used to measure “medium” characteristics. It is, however, essential that the threshold is capable of being overridden, based on modeler disagreement. Confinement, understandability, and maintainability are medium characteristics with respect to the quality model in Fig. 6.

“Weak” characteristics constitute the last type of this categorization and are similar to hunches. A hunch is something that seems to be wrong based on gut feeling, experience, or intuition. In fact, it is almost impossible to measure weak characteristics using metrics. At best, heuristics can support modelers, but they are unlikely to overrule experience or gut feeling. For example, purpose extraction can be checked partially by keyword comparison; however, the modeler has the last word.

A method for quickly and simply checking “weak” characteristics is simplified reviews. They separate different aspects of reviewing by altering a technique used in parallel thinking (Bono 1999). Each of the five review hats (cf. Fig. 8a) symbolizes a kind of thinking with clear goals and results in reviews no longer than necessary. It is important not to confuse the review hats with modeler expertise.

Without any ordering, each simple review can be conducted at any time. A yellow hat review (good points judgment) focuses on positive aspects of a model. It is assumed that a higher number of yellow reviews from different modelers describe a higher-quality model. For example, a yellow review might emphasize that a model is of great benefit in maintenance. When a model has a black hat review (bad points judgment), it possibly contains difficulties, dangers, defects, or a bad design. This type of simple review is commonly known as a “feedback or review.” Black hat reviews imply that an immediate patch of the model is needed. A white hat review (information) is used to provide or ask for information not directly available in the model. For instance, possible limitations due to modeler expertise are documented using white hat reviews. Focusing on creativity, green hat reviews (creativity) give the modeler feedback about possible improvements or new ideas. In model libraries, this review type is integral to fostering model evolution and to keeping modelers satisfied. Finally, general attitudes in terms of “like” and “dislike” are expressed by red hat reviews (emotions), e.g., dislike based on experience, with additional information regarding the reason, help improve the model.

We utilized simple reviews with review hats, as shown in Fig. 8b. The corresponding window allows access to the type of review and attachment of concerns or issues to the review. Additionally, review notes can be added (cf. the “Edit” button in Fig. 8b). The + next to the “Simple Reviews” in Fig. 4 creates a review.

“Strong”, “medium”, and “weak” characteristics are simple means to measure model quality characteristics. Integrating these measurement instruments into the staged model evolution approach allows continuous feedback from the modeler that influence design decisions and can be used to influence model evolution with respect to the underlying quality model.
5 Proactive model quality assessment

In the staged model evolution approach, a quality model is used to separate each stage, and quality gates verify that all quality characteristics of a stage hold true; however, quality checks are rarely done when they need to be triggered manually. This often leads to large error reports, which can be avoided if assessments are triggered automatically and, more importantly, periodically. Therefore, continuously triggered assessments analyze the model iteratively and provide feedback to the modeler. This changes the nature of quality feedback, making it more guided and avoiding large reports and suggestions for improvement. We refer to approaches that adhere to the aforementioned properties proactive quality guidance.

Proactive quality guidance is comprised of (1) automatic and continuous assessment of the current model and (2) clear instructions on the improvements. The primary focus is on automatic assessment when the model is changed and is based on the quality measurement instruments, as presented in Sect. 4.3. In general, manual approaches suffer from modeler obliviousness, i.e., the modeler forgets about triggering the assessment. Nevertheless, the challenge is to provide clear, simple, and instructive feedback. This method identifies areas of improvements and their cause and each time feedback is given, it must be correct. Otherwise, the modeler will neglect false feedback. However, the subjective nature of model quality makes it difficult to consistently derive correct feedback without manual interaction. The subsequent considerations narrow models to class diagrams:

In order to loosen the restriction requiring consistently correct feedback, the classification of quality measurement instruments, as described in Sect. 4.3, has been applied. Additionally, feedback is structured. While “strong” characteristics are the foundation for consistently correct feedback, “medium” characteristics are the basis for less precise feedback. Such feedback provides suggestions only and, therefore, can be neglected by the modeler. For instance, methods with long parameter lists should be avoided in order to limit what gets passed as a parameter. An excerpt of a list of metrics for “strong” and “medium” characteristics that can be used for feedback is shown in Tables 1 and 2. A more detailed list of possible metrics can be found in (Roth 2013). Lastly, with respect to the subjective nature of model quality, which is difficult to measure, simple reviews can be used to actively inform the modeler about the requirement to provide reviews. This is done

| Table 1 | Excerpt of model defects (Roth 2012) |
|---------|-----------------------------------|
| Model defect metrics | |
| Name | Description | References |
| AttrNameOvr | The 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 | Ramirez et al. (2004) |
| CyclicInheritance | Class inherits from itself directly or indirectly. The inheritance graph must be a tree; no cycles are allowed | Wüst (2014) |
| DupAttrNames | The class has two or more properties with identical names. Attribute names must be unique within the class | Wüst (2014) |
| DupOps | Class has duplicate operations. There are two or more operations with identical signatures (operation name and list of parameter types) | Wüst (2014) |
6 Approach assessment

We designed our model evolution approach for model libraries to meet one major goal and one major constraint (cf. Sect. 2). The goal was to find a precise and concise, yet simple and intuitive, foundation to explain model evolution in model libraries. The constraint was that our approach was meant to ease the production of model recommendations from these model libraries (cf. Fig. 2). Together, this reflects tool support in two respects: First, our tool support needs to offer a usability level that is as intuitive as possible, while supporting users. Specifically, it needs to be as self-explanatory as possible, while hiding the underlying concept. At the same time, it needs to enable as much automation as possible. This requires a sound and powerful conceptual foundation.

Second, our tool support is meant to ease the production of model recommendations. Therefore, simple quality statements about models are required in two respects. Quality statements should help the recommender algorithm rank models based on the assessed degree of reusability because “it is widely agreed that accurate predictions [recommendations] are crucial” (Ricci et al. 2011). Also, recommender systems that apply collaborative filtering mechanisms (Jannach et al. 2011) take user profiles into account in order to produce the best possible recommendations for a dedicated user. We use the level of experience as denoted in Sect. 4.1 as one cornerstone.

Altogether, we assess and discuss our approach subdivided into common ISO quality characteristics from the “system and software quality models”. Hence, we look at functional suitability, portability, usability, performance efficiency, and validity. Moreover, we discuss whether our approach might be more generally applicable than was reflected in our test environment.

Functional Suitability: To meet our requirements, we derived lessons learned from the literature (cf. Sect. 8) and designed a concept, transformed it into an approach, and created an implementation. This was carried out iteratively with paper prototypes in the beginning followed by a real implementation, as shown in Fig. 4 eventually. Using this development method enabled us to assess the feasibility of our concept using a software prototype (cf. Table 2

| Model smell metrics       | Description                                                                 | References |
|---------------------------|-----------------------------------------------------------------------------|------------|
| Long parameter list       | Do not pass everything the method needs; pass enough so that the method has everything it needs. | Fowler (1999) |
| Duplicates                | Duplicated attributes/methods/classes are bad                               | Fowler (1999) |
| Large class               | A class that is trying to do too much can be identified by determining how many instance variables it has. When a class has too many instance variables, duplicated code is likely involved | Fowler (1999) |
| Inappropriate intimacy    | Two classes are overly intertwined                                          | Fowler (1999) |

by updating the “Simple Review” section shown in Fig. 4. The modeler may use this feedback to improve the model.
This was possible due to our MDD, since feedback cycles from user studies could be injected into the prototype as feedback. Therefore, we omitted a further evaluation based on this aspect.

**Portability:** Our approach and the associated tool were developed mainly for UML class diagrams and, as a result, dependencies exist. For example, the metrics to check the quality attributes are UML specific (cf. Sect. 5). Otherwise, the concepts are generalized for many modeling languages. For example, one could consider the evolution basics, evolution graph, evolution stages, and evolution automaton, even though we did not evaluate it. This is unsurprising, as we show in our related work in Sect. 8. The quality model and the quality gates are likely general, as well, although properties like transformability may require refinement or discussion. For example, business process modeling notation (BPMN) models are often designed to be installed and activated in a business-process engine. One could argue that this requires a transformability, but this would differ from our format. While we aimed for model-to-text transformation, in a business-process environment, transformability would be of interest because it could be a BPMN interpreter executing the model immediately. If this holds true, then quality gates, quality characteristics, and simple reviews are generalizable. Therefore, automatic assessment and proactive quality is possible for BPMN models. Refinement of the metric suite is necessary for porting, keeping model reuse in mind. If the latter requirement is dropped, the approach might collapse, due to the quality model also requiring refinement.

**Usability:** We received feedback concerning usability, which allowed us to improve the overall user experience (Roth 2012). In a final usability study with ten participants, we tested our final implementation. Participants were randomly chosen computer science students with modeling backgrounds. Four of them were undergraduate students and six were graduate students. Their familiarity with modeling included participation in at least one UML modeling course, however, all still considered themselves beginners in UML modeling and none of them had a background in the related project. We addressed the following items in our user study: understandability of the stage colors, understandability of the stage transitions, identification of the current model stage, correctly understanding the trash bin, and whether the “pending” indicator is intuitive. All benchmarks were understood by 80 % of the participants, with details associated with the “pending” indicator.

![Fig. 9](image_url) Results of questions asked in the 10-steps task user study. (a) Staged model representation. (b) Quality measurement instruments.
indicator being the only item requiring additional explanation. (cf. Fig. 9a). Additionally, we assessed their understanding of concepts, including defects, smells, and reviews, and found that only smells required further explanation to 50% of the participants (cf. Fig. 9b). Defects and reviews were intuitive to all participants. As a consequence, we introduced tool tips specific for all of the above concepts, which provided further explanation.

We asked participants to perform a task using our tool. This task was divided into ten steps and participants were instructed to perform them individually. The steps were as follows: a model should be created and added to the model library, then the model should be edited until it reached the “fine” stage.

We evaluated participant understanding of the concepts during task performance. The questions were predefined and carefully designed to not guide or imply correct responses. The answers were used to evaluate the comprehensiveness of the presented concepts.

We also asked participants to evaluate how obtrusive the guidance in our approach was, with 90% considering it moderately obtrusive and 10% percent responding that the level was low. We interpret this result as indicating that the guidance was present enough to get noticed, while not being annoying, which was our desired response.

**Performance Efficiency:** The proactive nature of our approach requires calculation of several metrics during editing. This becomes computationally complex as the model size increases. Given our deployment scenario involving model libraries with models of limited size, this was not an issue. Even using an office computer, specifically a Lenovo ThinkPad T410s (Intel Core i5-540M, 4 GB RAM, DDR3-1330) released in 2010, the continuous calculations were instantaneous. Consequently, we can assure that assessments from Sect. 5 work in almost real-time, however, some metrics are computationally expensive, meaning that the graph planarity is NP-hard (Garg and Tamassia 1995). Hence, for larger models, such as industrial-sized models with several thousands of elements, we cannot promise a real-time user experience. Otherwise, it is feasible to store large models on model servers and distribute the metrics calculations. Furthermore, algorithms and libraries continuously improve, such as EMF IncQuery consistently optimizing queries and execution (Ujhelyi et al. 2015). This does not alter the presented approach, however, we did not perform any tests to specifically address these possibilities.

**Validity:** Evaluations inherit shortcomings and often provide weak evidence (Segal 2003). Hence, we tried to overcome some threats by carefully designing and testing (pre-test) each question in a pre-planning. This lead to rephrasing each question several times to address biased answers, i.e., Hawthorne and Rosenthal effects, which are well known in social sciences. For the test itself, we opted against a double-blind test to rule them out perfectly. This may result in biased answers though we specifically asked for honest answers. Concerning our participants (selection-bias), we chose only volunteers with a minor background in modeling and consider them as a baseline of possible users. Our assumption is that more experienced modelers will perform and understand the approach and tool better. Moreover, we ensured an all equal environment for every test and participant (situational effects). Besides that, the tool itself evaluates the approach and demonstrates that it meets the given requirements. This in itself imposes threats to validity because our focus lies in UML and often class diagrams. A change to entity relationship, AUTOSAR, BPMN, or software architectural models will require adjustments to “strong” and “medium” quality measurement instruments or even more, but we did not investigate on that or claim that our approach is so general. While, provided the proper metrics, adjustments to entity relationship models seem easy, our experience in AUTOSAR is just not sufficient to make a statement. Finally, our conducted experiment is only one step to evaluate our approach and tool, but results are promising. Nevertheless, a long-term study
is required, given that we cannot possibly anticipate all of the potential threads why our approach and tool might fail in the future. At the moment, however, we cannot anticipate reasons why our approach and tool would not be successful. All in all, we tried to address the common internal and external threads by focusing more on quality design than on quantity of participants.

Fig. 10  Extended model evolution with generations
We showed that our approach is concise, precise, and sufficiently automated in a software implementation, as explained in Sect. 5 (cf. Ganser 2014c, a). Moreover, we found that our model library now has an easy platform for evolving and discussing models, which is accepted by modelers. The benefits of higher-quality models for a model recommender are twofold. First, simple filtering can omit immature models and transform others into recommendations, given that novice users might not be able to handle incorrect models. Second, the quality can be taken into account for ranking models from recommender systems. Still, if no model is found that qualifies as a recommendation, it is possible to lower quality models and inform the modeler. This is a different paradigm as compared to recommender systems, which strive for user satisfaction by “providing best prediction accuracy at minimal distraction to the user.” (Golbandi et al. 2010; Ricci et al. 2011).

7 Extended model evolution approach

The model evolution approach for model libraries presented thus far considers individual models, however, our model library stores interlinked models (Ganser and Lichter 2013). We now elaborate on how our model library forms an enhanced knowledge graph and explain how this can be used to extend our evolution approach to provide data to produce more accurate recommendations.

7.1 Model libraries as graphs

Our model library was designed to both store models and offer functionality to relate them. To this end, we used a graph structure where every model was represented by a vertex, allowing the creation of edges between vertices to represent certain semantics (Ganser and Lichter 2013). This enabled a UML class diagram to be split into two diagrams with one remaining relationship between them. Therefore, if these two diagrams are stored as vertices, the relationships can be stored as an edge. We call an edge containing syntactical information a cross-link. In our generation example in Fig. 10, the Airport model could be a vertex, the Passenger model another vertex, and a link between these two vertices could hold an association between the classes Airport and Passenger. Altogether, vertices and edges constitute an enhanced knowledge graph, which uses hyperedges, because often more than two vertices are linked.

The benefit gained from models embedded in an enhanced knowledge graph is the ability to produce chain recommendations. These recommendations can be one of three types. First, related models can be chosen together with the recommended model. In our example, a chosen Airport model would also imply a chosen Passenger model, allowing creation of all the classes on the canvas at once. Second, models can be recommended as follow-ups after the recommended model was chosen. For example, the recommendation list produced after choosing the Airport model puts the Passenger model on top of the list without any further querying. Finally, these models can be assessed differently, or higher ranked, in the next round of producing recommendations. This means a chosen Airport model leads to a higher ranking for the Passenger and Vehicle models in the next round of producing a recommendation list.

Since models can be linked in our model library while evolving independently, the question becomes what happens to the links if one of the models evolves while the other...
does not? Rather than a question of co-evolution, this requires considering ripple effects at the syntactic level of links ignoring the semantic and the generic level of links (Ganser and Lichter 2013).

7.2 Generations of evolving models

Considering our Airport example from Fig. 3, we can extend this to explain generations. This concept is similar to what generations refer to in biology. Here, it represents an evolving group of linked snapshots. For example, Fig. 10 places the known sequence of snapshots in the middle and displays one model above and one below. Additionally, three snapshots, one in each row, are printed, readable, and grouped with grey backgrounds. They are grouped because they are linked on a syntactical level. For instance, one association between the Airport class is established with the Passenger class and another association is established between the Airport class and the Vehicle class. The three models are independent and syntactically correct models in our model library, because the information about the relationships is stored in the edges of the enhanced knowledge graph. Still, this group of vertices and edges, each represented by a snapshot, form a generation in our model library.

The first question now to be answered, when considering generations in this model library, is how to combine it with the staged model evolution approach. Every model of a generation might be in a different stage, so in Fig. 10, our Airport model might be in stage “decent”, while the Passenger and the Vehicle models are in the “vague” stage. Bearing generations as a means for supporting chain recommendations, we define the stage of a generation as the lowest of all contained models, i.e., “vague”.

In a case where one of the models in a generation is marked as deprecated, the entire generation is considered deprecated. However, the impact on chain recommendations is only affected in relation to the deprecated model, while every other model is still able to contribute in a chain recommendation. For example, the generation from above would be handled exactly as described in the chosen example, even if it contained a deprecated Vehicle model.

7.3 Impact analysis and change propagation

Discussing a possible impact of changes to a model in our model library, we need to keep in mind that we have an enhanced knowledge graph at hand, meaning that we have no upgrade conflicts or merge conflicts (Mens and D’Hondt 2000). Therefore, changes made to a model in our library are not forwarded to reused models. Moreover, we lock models and avoid getting two parallel versions of the same model, which would need to be merged.

Hence, two structural and compositional conflicts remain to be addressed (Mens and D’Hondt 2000). The first would be a syntactically incorrect model that impacts another model, however, this is dealt with by the syntactic aspects of our quality guidance. The second, the compositional conflicts, occur if the composition of a generation changes its semantics, which is dealt with by the semantics of our quality guidance. These two conflicts are referred to as syntactic consistency and semantic consistency (Mens et al. 2005).

We can now look at the impact certain types of model changes might have. A common approach classifies changes as addition, removal, connection, and disconnection (Mens and D’Hondt 2000; Mens et al. 2005). Moreover, grouping mechanisms, such as promotion and sequence mentioned above mentioned, are supported. While the presented approach
extends the UML meta-model to realize “evolution contracts”, we continue to use the terminology.

This terminology leads to the following observations. Considering element “addition” in our model library, no harm can be done. The worst that can happen is a duplicate element in an adjacent model that is linked to the changed model via cross-link. However, this is only a potential conflict, which only needs to be resolved as both models are used in the same model. This is something the reuse merger addresses by prefixing one of the elements. The same applies for the “connection”. Next, the “disconnect” cannot harm the model in a conflict sense, however, it can introduce model smell. This might happen if a model gets partitioned into two unrelated parts, implying that they should be held as two models in our model library. Finally, “removal” might lead to real conflicts.

These “removal” conflicts are approached in two steps: first a change-impact analysis needs to be undertaken (Bohner and Arnold 1996). This is followed by change propagation, which checks to see if a change in one model induces changes in a cross-link or if changes in another model need to be undertaken (Vclav Rajlich 1997).

Algorithm 1 Impact Analysis

1: function IMPACTANALYSIS(model, crossLink)
2:     adjacentModel ← GETADJACENTMODEL(model, crossLink)
3:     ASSERTSYNTAXOK(model, adjacentModel, crossLink) ▷ Something wrong before
4:     mergedModel ← MERGE(model, adjacentModel, crossLink)
5:     if isValid(mergedModel) then return ▷ No harm done
6:     leftLinks ← MODELSETDIFFERENCE(crossLink, model)
7:     if ISNOTEMPTY(leftLinks) then return leftLinks ▷ Issues found in links to model
8:     rightLinks ← MODELSETDIFFERENCE(crossLink, adjacentModel)
9:     if ISNOTEMPTY(rightLinks) then return rightLinks ▷ Issues in links to adjacent
10:    return emptySet

The first step (impact analysis) works as follows for each cross-link (cf. algorithm 1): assuming that the changed model and the adjacent model are syntactically correct, we first do a merge test that attempts to merge the changed model, one cross-link, and the adjacent model. If this result is syntactically correct, no harm was done. If this results in a syntactically defective model, the issue lies either in the cross-link or in the adjacent model. To find out, we first build the set difference of the cross-link concerned with the changed model and the changed model itself (called MODELSETDIFFERENCE in algorithm 1). If this results in an empty set, the issue must exist in the adjacent model. Hence, we build the set difference of the adjacent model and the cross-link concerned with the adjacent model. This cannot result in an empty set unless the sources were syntactically incorrect, which is only possible if the adjacent model was in a “vague” stage. Moreover, this check can be done proactively, because the cross-links are always available and each change in a model can be found in the cross-link. We avoid doing this, because models in the “vague” stage might not be well formed.

In the second step (change propagation), we need to let the modeler decide. This is due to the semantic impact that a change might have, however, the results from the impact analysis can be used by the modeler to fix discovered conflicts. For example, in our Airport model the class Airport was renamed. This resulted in a conflict between the Airport model and the cross-link, because the cross-link points to a class named Airport. Fortunately, this conflict is easily resolved by adjusting the cross-link. In fact, matchings with identifiers are possible, however, we want to keep the example simple. Even in more complex examples, automatic conflict resolution should be manageable,
because cross-links do not comprise many references to models and the checks can be done proactively.

We implemented an extension of our evolution approach for models offering an extended perspective on models in our model library and providing additional data that a recommender can take into account. Hence, the concept belongs in the domain of model evolution, however, the evaluation is a part of the benchmarking recommender algorithm and remains in progress.

8 Related and complementary work

We first match our approach to common classification criteria in order to ease contrasting our own contribution to that of others. We do so by looking at, “the four different dimensions of evolution in MDD” (van Deursen et al. 2007). Next, we consider “the two orthogonal model evolution dimensions” (Biehl 2010). Finally, we look at “the three types of changes in software model evolution” (Levendovszky et al. 2011).

First, Van Deursen et al. provided four dimensions of evolution in MDD and our approach falls into regular evolution (van Deursen et al. 2007). One might think that our approach could also fall into abstract evolution, however, the model libraries we are considering are not meant to be linked to the reused models. Hence, we consider the models detached as soon as they are reused, because we found that models too often need to be altered in their reusing environment. This renders change propagation in this direction pointless.

Second, concerning Biehl’s two orthogonal dimensions of model evolution, our approach involves content-related changes and local evolution (Biehl 2010). This is because we concentrate on add, delete, rename, and retype operations and leave the abstract syntax untouched. Also, we isolate ripple effects and do not automatically propagate changes between interlinked models.

Last, the change types, according to Levenovszky et al., are requirements and style (Levendovszky et al. 2011). The first is due to a purpose that might not be in line with the model and the second might be used to enhance comprehensibility. Additionally, corrective or perfective changes might happen, as well (Swanson 1976), however, the modeling language itself remains unchanged. This is accounted for by our prototype environment that builds upon EMF/Ecore (Steinberg et al. 2008).

With the classifications set for model evolution dimensions and change types, our approach can be closely related to model-change propagation, model libraries, and model quality. Hence, we present the current understanding of these areas and limit the scope to model evolution in model libraries. We then contrast the differences between the current understanding of model evolution and our method.

8.1 Model evolution

The continuous change of software, which is often seen as evolutionary behavior, has been observed and characterized by Lehman (1980). They investigated software development over 30 years and presented the results as a set of laws. These Lehman laws are considered general observations rather than laws of nature. They pertain to all systems that model real-world processes and become an indisputable part of the real world.
Model evolution is often investigated as a goal to be achieved automatically by software-driven tool support. There are several tools and research prototypes currently available. First, COPE supports evolution and co-evolution by monitoring changes in an operation-based way (Herrmannsdörfer 2011). These can be applied as editing traces to other models (Herrmannsdörfer and Ratiu 2010). Moreover, these traces can be stored in a library in order to be forwarded or applied to other models, as well. This is important, as co-evolution is the focus of this project. Our approach differs in that we do not trace changes, but instead focus on edit-time changes and their impact on quality aspects. However, some similarities between COPE and our approach can be found in the stages that underlie the approach. In the end, we need more formally defined stages, while COPE needs more formally defined operations. These operations are derived from object-oriented database-schema experience and object-oriented source-code experience (refactoring) and are an overview of high-level operations. Altogether, operations are seen as structural or non-structural primitives. For example, specialization, generalization, delegation, and inheritance are such primitive operations, while others include replacement and mergesplit (Herrmannsdörfer et al. 2011). Furthermore, non-structural primitives are considered (“make class abstract”), which leads to a more subtle difference, because our retype is model inherent. This means that a class does not become an interface, but is deleted and then created as one. Hence, retype considers type changes of attributes.

Next, MoDisco (Eclipse Foundation 2014b) [hosted with AM3 (Allilaire et al. 2006)], provides the means to support the evolution of legacy systems by applying model-driven ideas. This means that MoDisco is a tool for re-engineering legacy software by means of models and starting a MDD from derived models. Additionally, co-evolution is taken into account. Our focus is on plain model evolution, with the main distinction of our approach being that we want evolution to be guided and directed instead of aimless.

Another approach follows the idea of tracing model elements in version histories (Wenzel et al. 2007). This enables monitoring of model changes over time in order to find commonalities and differences. The authors do not mean to put their work in the context of model evolution, but in our approach, this enables finding operation sets as they occur between snapshots. What is missing, relative to our approach, are concepts, including stages or quality gates, which add semantic meaning to sequences of snapshots in terms of reusability.

Finally, model evolution can be achieved using model transformations by demonstration (Sun et al. 2009). These demonstrations are meant to assist end-users and automate model evolution tasks (Taylor et al. 2011). This means that the underlying concepts and ideas differ tremendously and that users of the tools are totally different. While MT-Scribe, the tool behind this approach, targets end users, we aim at modeling experts.

8.2 Model quality

Our evolution approach was enhanced using ideas concerning quality in modeling by Moody (2005), because we wanted to establish a common understanding of model quality in our library and avoid the expectation that, “quality is seen as a subjective and rather social than a formally definable property lacking a common understanding and standards” (Moody 2005). This is possible, because model libraries noticeably limit the scope, therefore, we were able to develop an approach to measure model quality in model libraries. Hence, we used the quality dimensions described by Lindland et al. (1994) and applied them to our environment. They comprise syntactic, semantic, and pragmatic
quality, bearing in mind that these quality dimensions can influence each other, as presented by Bansiya and Davis (2002).

Considering UML models, there exist manifold model qualities. We chose the work of Lange (2006) to be most suitable and linked it with metrics, keeping in mind the work of McQuillan and Power (2006), and Kim and Boldyreff (2002). We then employed the work of Genero et al. (2003), (Wedemeijer 2001), and (Mohagheghi and Dehlen 2009), who described model quality as a set of dimensions and identified two primary cases of model use: maintenance and development. For each of these, model characteristics that influence multiple model purposes are derived from the proposed set of dimensions. The characteristics have to be measured to draw conclusions about model quality. However, Mohagheghi et al. point out that measuring the presented model characteristics is challenging (Mohagheghi and Dehlen 2009). These challenges are greatly reduced when model evolution is considered for model libraries, since model quality can be limited to general qualities. Furthermore, we needed a means to assess some semantic and pragmatic aspects. Here we root our ideas on reviews, but Fagan’s approach is too complex (Fagan 1976). Hence, we subdivided review tasks using an adaption of the Six Thinking Hats, as proposed by De (Bono 1999).

With respect to our measurements, a tool called EMF Refactor is the most similar tool currently available (Arendt et al. 2010a). The name is slightly misleading in our context, however, one part of was formerly referred to as EMF Metrics (Arendt et al. 2010b), and was later joined to that project (Arendt and Taentzer 2013). The most important aspects discussed in this context are the model metrics and smells (Arendt and Taentzer 2012), which are the foundations for model refactoring. However, the framework could serve as a metrics calculation engine in our environment as well. The difference in our approach is that we added hunches to the concept and put a stronger focus on edit-time assessments.

A tool called MetricViewEvolution is possibly the most similar tool described in current literature (Lange et al. 2007a, 2007b). It offers six different views of evolved models, namely, context, meta, metric, UML-city, quality, and evolution. To that end, the description is very user-interface-centric and hides conceptual details. Our approach is more centered on concepts and lightweight, simple user interfaces.

8.3 Model repositories and libraries

The goals and functionality of model repositories allow for querying, conflict resolution, and version management, but little more. This means that evolution and co-evolution are not considered. An example for a model repository that primarily focuses on functionality as defined from source-code version-control systems is the Adaptable Model Repository (Altmanninger et al. 2008). This realizes version control for models, as well as operation- and semantic-based conflict detection and resolution. This is accomplished by overcoming the drawbacks of XMI-serialization. The second example is the Repository for Model-driven Development (ReMoDD), a model repository focusing on community building by offering models to the modeling community (France et al. 2006). It is meant to be a library that contains examples for case studies or educational purposes. Consequently, ReMoDD does not support reuse for modeling out of the box. This is supported by the fact that many of the models are stored as screenshots or in PDF files, i.e., lack a machine-readable format. This hinders modelers from reusing models quickly, because they need to recreate each and every artifact. A third example is the “Model Repository” developed at The University Leipzig (Elinson et al. 2010). It uses a combination of EMF Eclipse technologies and the Neo4j graph databases for storing models. To that end, it relies on the
Apache Lucene Engine shipped with Neo4j. The fourth example is a user-friendly UML model repository developed by France Telecom (Belaunde 1999). It uses a layered architecture similar to the modeling levels and enables remote model storage. In that respect, it stores not only (meta) models but also instances and functions, as one would expect for a repository. This means that reuse is not a focus. Another model repository that does not focus on model reuse is EMFStore (Kögel and Helming 2010; Kögel 2011), which supports collaborative work on models. The general idea is to store models centrally for collaborative work and enable model exchange between different modelers as if they were under version control. Moreover, changes are propagated as they are stored.

In contrast to model repositories, MOOGLE aims at model reuse and builds on top of model repositories (Lurêdio et al. 2012). It is a user-friendly model search engine offering enhanced querying by creating a rich index. Hence, models in XMI format are preprocessed and indexed by an Apache Lucene engine. This enables fuzzy searches, rankings, browsing, previews, and assistance for finding unrelated, but relevant models.

A community-based model repository aimed at “development, analysis, and reuse” is MDEForge (Basciani et al. 2014), which combines the concepts of ReMoDD and MOOGLE to a degree. Furthermore, it is meant to enable the adoption of model-management tools to software-as-a-service. Therefore, the meta-model is kept simple and extendable.

Another meta-model-based approach with a wider area of application is megamodeling (Bézivin et al. 2005). The goal is to adapt the idea of programming-in-the-large, implemented with a module interconnection language (MIL), as proposed by DeRemer and Kron (DeRemer and Kron 1975), to modeling and create modeling-in-the-large. This means that a distinguished model is meant to play the role of the MIL and provide descriptions on a meta level. Specifically, these descriptions are relationships that can exist between models and other project-related documents. In a sense, this model holds together all the relationships between artifacts in a project and enables traceability. One of the major claims in megamodeling is that model transformations and model weaving are key for modeling environments (Allilaire et al. 2006).

On a more formal level, there are approaches similar to our enhanced knowledge graph. For example, Voigt follows the idea of separating models for the purpose of model matching (Konrad Voigt 2011). Strüber et al. also split a model into smaller parts, however, they did that for modularity purposes and applied crawling techniques (Struber et al. 2014).

Contrary to the above mentioned model repositories that offer version control, at best, a model library attempts to archive models rather than store them. As a result, models in a model library need to be stored in a well-organized manner. This is why we developed a model library that stores models in an enhanced knowledge graph (Ganser and Lichter 2013). Here, a model is represented as a vertex and can be related to other vertices (models by cross-links) (cf. Sect. 7.1) which offer much more detail and data than the other approaches discussed. Our approach forms an enhanced knowledge graph that enables a recommender framework to produce model and chain recommendations (Dyck et al. 2014a).

8.4 Model-change propagation

Our approach to generations deals with ripple effects differently compared to what has been studied as model-change propagation, model-consistency management, or model synchronization. This is because we aim at evolution support for model libraries with interlinked models. Similar to Levendovszky et al., we are considering intramodel and
intermodel conditions, though we do not keep to one single domain (Levendovszky et al. 2011).

The first related approach uses evolution contracts (Mens and D’Hondt 2000). This idea is derived from reuse contracts and realized as an extension to the UML meta-model. This allows change tracking and guide evolution to be more disciplined. Here a “framework for managing consistency of evolving UML models” comes in handy (Mens et al. 2005), because it supports impact analysis, consistency verification, and change propagation. In other words, it assures “safe model evolution”. In our environment, an extension of the underlying meta-model (Ecore) is not worth striving for.

Another related approach by Dantas et al. uses model-mining techniques to detect change traces in a model version-control system (Dantas et al. 2005). These are meant to support impact analysis activities. To do so, this approach builds on a tool called Odyssey-VCS (Murta et al. 2007), which is a tool that applies a mechanism used in source-code version-control systems to the model level. Moreover, Odyssey-VCS supports different diagram types and link traces. Our approach works differently, because it can monitor changes during edit time. Consequently, there is no postprocessing like model-mining necessary.

Briand et al. presented an approach for “automated impact analysis of UML models” (Briand et al. 2006). It proposed a change taxonomy, a methodological framework, and used object constraint language (OCL) in its realization. Hence, the whole approach relied on a formal foundation regarding model changes, their impact, and sets of impacted elements they called “bag of impacted elements”. Additionally, they defined a distance measure between changed and impacted elements. We used ideas and concepts from their approach, but did not automate.

Mens et al. studied “consistency between UML models” and employed description logics (Mens et al. 2003). Therefore, they used XMI exports and transformed them into description logics. Finally, they distinguished between horizontal consistency, which is consistency between models with the same version information, and evolution consistency, which denotes consistency between versions of one model. In some respect, the horizontal consistency is similar to our generation, but we did not require our models to be of the same version.

Another field we took into account for our work is called model refactoring. Tools are often referred to as refactoring browsers and an introduction and overview was provided by Boger et al. (Boger et al. 2003). An approach employing openArchitectureWare/OCL queries is described by Enckevort (van Enckevort 2009). Some larger scale refactoring was proposed by Astels (Astels 2002), who relied on code smells and escalated changes to model level. Moreover, he briefly discussed “refactor to patterns”. Finally, an operation based approach is the “Operation Recorder” (Brosch et al. 2009), which subdivides refactorings into eight steps, from “create initial model” to “generate specific artifacts”, that are explained in a state machine example. All in all, for our approach, these ideas proved beneficial, because refactorings often struggle with the same issues as we do with generations, such as conflict detection and impact on static-versus-dynamic views.
9 Summary and conclusion

General model evolution is to be distinguished from model evolution in model libraries as briefly discussed above. This is due to unguided evolution being self-defeating for model libraries. Given this, guidance is required to keep models reusable. Furthermore, a model library focused on reuse puts constraints on a quality model for models that make it manageable.

We have shown how models should evolve in model libraries with proactive quality guidance (cf. Fig. 11) and illustrated how model evolution in model libraries can be described in steps and snapshots, as well as how a quality model for a model library can be used to guide and stage model evolution in model libraries. To achieve this, we broke down the stages to “vague”, “decent”, and “fine” quality characteristics, which are verified in different ways. While “strong” characteristics are checked automatically, “medium” and “weak” characteristics require user interaction. However, this interaction is supported in two ways: (1) for “medium” characteristics, some metrics provide assessments that only need to be judged by a user, given that certain thresholds might not hold true for a particular case; (2) for “weak” characteristics, we introduced simple reviews that allow quick and guided evaluations. Finally, we extended the approach to groups of models in model libraries we call generations. These are of particular importance for reuse in the context of model recommenders.

Since all of this takes place during editing time, we call this approach proactive quality guidance. Since existing metrics allow suggestions on how to fix certain issues, we created a prototype that implements the entire approach (Ganser 2014c). This prototype looks simple and clean and avoids as much noise for modelers as possible in order to avoid modeler distraction while modeling.

Acknowledgments We would like to thank our reviewers and editors for their great contributions. It’s been a long journey from our first submission. On this way, we received incredible feedback that improved the quality of the article tremendously and discussions unveiled new and very interesting research directions. We would also like to thank all our co-workers for their contributions and patience in endless discussions. Last, but not least, big thanks to all our participants in our tests, surveys and studies.
References

Allilaire, F., Bézivin, J., Bruneliere, H., & Jouault, F. (2006). Global model management in eclipse GMT/AM3. In D. Thomas (Ed.), Proceedings of the eclipse technology exchange workshop (eTX) at the ECOOP 2006 conference, Lecture notes in computer science (Vol. 4067).

Altmanninger, K., Kappel, G., Kusel, A., Retschitzegger, W., Seidl, M., Schwinger, W., et al. (2008). AMOR—Towards adaptable model versioning. In 1st international workshop on model co-evolution and consistency management (MCCM’08), workshop at MODELS’08, ACM, pp. 1–7.

Arendt, T., & Taentzer, G. (2012). Integration of smells and refactorings within the eclipse modeling framework. In Fifth workshop on refactoring tools (WRT 2012), ACM, New York, NY.

Arendt, T., & Taentzer, G. (2013). A tool environment for quality assurance based on the Eclipse Modeling Framework. Automated Software Engineering, 20(2), 141–184. doi:10.1016/j.ase.2011.05.008.

Arendt, T., Manzt, F., & Taentzer, G. (2010a). EMF refactor specification and application of model refactorings within the Eclipse Modeling Framework. In Proceedings of 9th BElgian-NEv erlands software eVOlution seminar.

Belaunde, M. (1999). A pragmatic approach for building a user-friendly and flexible UML model repository. In R. B. France, B. Rumpe (Eds.), UML’99—The Unified Modeling Language, Lecture notes in computer science (Vol. 1723, pp. 188–203). Berlin: Springer. doi:10.1007/3-540-46852-8_14.

Bézivin, J., Jouault, F., Rosenthal, P., & Valduriez, P. (2005). Modeling in the large and modeling in the small. In U. Allmann, M. Aksit, A. Rensink (Eds.), Model driven architecture, Lecture notes in computer science (Vol. 3599, pp. 33–46). Berlin: Springer. doi:10.1007/11538097_3.

Biehl, M. (2010). Supporting model evolution in model-driven development of automotive embedded systems. Licentiate Thesis, KTH, Stockholm, Sweden.

Boger, M., Sturm, T., & Fragemann, P. (2003). Refactoring browser for UML. In Objects, components, architectures, services, and applications for a networked world, Lecture notes in computer science (Vol. 2591, pp. 366–377). doi:10.1007/3-540-36557-5_26.

Bohmer, S. A., & Arnold, R. S. (1996). Software change impact analysis. Los Alamitos, CA: IEEE Computer Society Press.

Bono, E. (1999). Six thinking hats: Run better meetings, make faster decisions. In Six thinking hats. London: Penguin.

Briand, L. C., Labiche, Y., O’Sullivan, L., & Sówka, M. M. (2006). Automated impact analysis of UML models. Journal of Systems and Software, 79(3), 339–352. doi:10.1016/j.jss.2005.05.001.

Brosch, P., Langer, P., Seidl, M., Wieland, K., Winmer, M., Kappel, G., et al. (2009). An example Is worth a thousand words: Composite operation modeling by-example. In A. Schürr, & B. Selic (Ed.), Model driven engineering languages and systems, Lecture notes in computer science (Vol. 5795, pp. 271–285). Berlin: Springer. doi:10.1007/978-3-642-04425-0_20.

Chaudron, M., Heijstek, W., & Nugroho, A. (2012). How effective is UML modeling? Software & Systems Modeling, 11(4), 571–580. doi:10.1007/s10270-012-0278-4.

Dantas, C.R., Murta, L.G., & Werner, C. (2005). Consistent evolution of UML models by automatic detection of change traces. In Proceedings of the 8th international workshop on principles of software evolution, IEEE (pp. 144–147). doi:10.1109/IWPSE.2005.10.

DeRemer, F., & Kron, H. (1975). Programming-in-the-large versus programming-in-the-small. ACM SIGPLAN Notices, 10(6), 114–121. doi:10.1145/390016.808431.

Dreyfus, S. E., & Dreyfus, H. L. (1980). A five-stage model of the mental activities involved in directed skill acquisition. http://goo.gl/76GVH8.

Dyck, A., Ganser, A., & Lichter, H. (2013). Enabling model recommenders for command-enabled editors. In MoDELS MDEBE—International workshop on model-driven engineering by example 2013 co-located with MODELS conference, September 29, 2013 (pp. 12–21). Miami FL: CEUR.

Dyck, A., Ganser, A., & Lichter, H. (2014a). 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 (pp. 282–290). doi:10.5220/0004701702820290.
Dyck, A., Ganser, A., & Lichter, H. (2014b). 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 (pp. 291–299). doi:10.5220/0004701802910299.

Eclipse Foundation. (2014a). Code recommenders. http://www.eclipse.org/recommenders/.

Eclipse Foundation. (2014b). MoDisco. http://www.eclipse.org/MoDisco/.

Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., & Mockus, A. (2001). Does code decay? Assessing the evidence from change management data. IEEE Transactions on Software Engineering, 27(1), 1–12. doi:10.1109/32.895984.

Elinson, S., Hanns, M., Kirstein, M., Kronseder, S., & Köhler, S. (2010). Eclipse model repository. http://hermes.modelrecommenders.org/#.

Embley, D. W., & Thalheim, B. (Eds.). (2011). Handbook of conceptual modeling: Theory, practice, and research challenges. Berlin [u.a.]: Springer.

Fagan, M. E. (1976). Design and code Inspections to reduce errors in program development. In IBM systems journal (Vol. 15, IBM, pp. 182–211). doi:10.1147/sj.153.0182.

Fowler, M. (1999). Refactoring: Improving the design of existing code. Reading, MA: The Addison-Wesley Object Technology Series, Addison-Wesley.

France, R. B., Bieman, J., & Cheng, B. H. C. (2006). Repository for model driven development (ReMoDD). In T. Kühne (Ed.), Proceedings of the 9th international conference on model driven engineering languages and systems, ACM, Lecture notes in computer science (Vol. 4364, pp. 311–317). doi:10.1007/978-3-540-69489-2_38.

Ganser, A. (2014a). The HERMES project—Demo videos. http://hermes.modelrecommenders.org/demos.html.

Ganser, A. (2014b). The HERMES project—Eclipse P2 Updatesite: HERMES.reuse. http://goo.gl/ZGxIf.

Ganser, A. (2014c). The HERMES project: Harvest, evolve, and reuse models easily and seamlessly. http://hermes.modelrecommenders.org/.

Ganser, A., & Lichter, H. (2013). 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 (pp. 135–142). doi:10.5220/0004320801350142.

Garg, A., & Tamassia, R. (1995). On the computational complexity of upward and rectilinear planarity testing. In G. Goos, J. Hartmanis, J. Leeuwen, R. Tamassia, & I. G. Tollis (Eds.), Graph drawing, Lecture notes in computer science (Vol. 894, pp. 286–297). Berlin: Springer. doi:10.1007/3-540-58950-3_384.

Genero, M., Piattini, M., & Calero, C. (2002). Empirical validation of class diagram metrics. In International symposium in empirical software engineering, Kluwer Academic Publishers, Empirical Software Engineering (Vol. 7, pp. 195–203). doi:10.1109/ISESE.2002.1166940.

Genero, M., Piattini, M., & Calero, C. (2002). Empirical validation of class diagram metrics. In International symposium in empirical software engineering, , Kluwer Academic Publishers, Empirical Software Engineering (Vol. 7, pp. 195–203). doi:10.1109/ISESE.2002.1166940.

Golbandi, N., Koren, Y., & Lempel, R. (2010). On bootstrapping recommender systems. In J. Huang (Ed.), CIKM ’10, ACM, [New York] (pp. 1805–1808). doi:10.1145/1871437.1871734.

Herrmannsdörfer, M. (2011). Evolutionary metamodeling. PhD: Technische Universität München, München.

Herrmannsdörfer, M., & Ratiu, D. (2010). Limitations of automating model migration in response to metamodel adaptation. In S. Ghosh (Ed.), Models in software engineering, Lecture notes in computer science (Vol. 6002, pp. 205–219). Springer. doi:10.1007/978-3-642-12261-3_20.

Herrmannsdörfer, M., Vermolen, S. D., & Wachsmuth, G. (2011). An extensive catalog of operators for the coupled evolution of metamodels and models. In B. Malloy, S. Staab, & van den Brand, M. (Ed.), Third international conference software language engineering (SLE 2010), Lecture notes in computer science (Vol. 6563, pp. 163–182). Berlin: Springer. doi:10.1007/978-3-642-19440-5_10.

Jannach, D., Zanker, M., Felfernig, A., & Friedich, G. (2011). Recommender systems: An introduction. New York: Cambridge University Press.

Judson, S. R., France, R. B., & Carver, D. L. (2004). Supporting rigorous evolution of UML models. In Proceedings of the ninth IEEE international conference on engineering complex computer systems 2004, IEEE, pp. 128–137. doi:10.1109/ICECCS.2004.1310911.

Keenbul, F., & Rausch, A. (2001). Using XML/XMI for tool supported evolution of UML models. In Proceedings of the 34th annual Hawaii international conference on system sciences, IEEE, p. 10. doi:10.1109/HICSS.2001.927259.
Kim, H., & Boldyreff, C. (2002). Developing software metrics applicable to UML models. In G. Goos, J. Hartmanis, J. van Leeuwen, J. Hernández, & A. Moreira (Eds.), Object-oriented technology ECOOP 2002 workshop reader (Vol. 2548). Berlin: Springer.

Kögel, M. (2011). Operation-based model evolution (1st ed.). München: Verlag Dr. Hut.

Kögel, M., & Helming, J. (2010). EMFStore: A model repository for EMF models. In J. Kramer (Ed.), Proceedings of the 32nd ACM IEEE international conference on software engineering, ACM, New York, NY, p. 307. doi:10.1145/1810295.1810364.

Lange, C. (2006). Improving the quality of UML models in practice. In 28th international conference on software engineering, association for computing machinery (pp. 993–996). New York, NY. doi: 10.1145/1134285.1134472.

Lange, C., & Chaudron, M. (2005). Managing model quality in UML-based software development. In International workshop on software technology and engineering practice (Vol. 0, pp. 7–16). doi: 10.1109/STEP.2005.16.

Lange, C., Wijns, M. A., & Chaudron, M. (2007a). A visualization framework for task-oriented modeling Using UML. In 40th Hawaii international conference on system sciences. IEEE, Piscataway, NJ, p 289a. doi:10.1109/HICSS.2007.44.

Lange, C., Wijns, M. A., & Chaudron, M. (2007b). MetricViewEvolution: UML-based views for monitoring model evolution and quality. In Krikhaar, R. (Ed.), IEEE computer society 11th European conference on software maintenance and reengineering, Los Alamitos, CA. [u.a.], pp. 327–328. doi:10.1109/CSMR.2007.32.

Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. Proceedings of the IEEE, 68(9), 1060–1076. doi:10.1109/PROC.1980.11805.

Levendovszky, T., Rumpe, B., Schätz, B., & Sprinkle, J. (2011). Model evolution and management. In H. Giese, G. Karsai, E. Lee, B. Rumpe, & B. Schätz (Ed.), Model-based engineering of embedded real-time systems, Lecture notes in computer science (Vol. 6100, pp. 241–270). Berlin: Springer. doi: 10.1007/978-3-642-16277-0_9.

Lindland, O. I., Sindre, G., & Solvberg, A. (1994). Understanding quality in conceptual modeling. IEEE Software, 11(2), 42–49. doi:10.1109/52.268955.

Lucrédio, D., Fortes, M., Renata, P., & Whittle, J. (2012). MOOGLE: A metamodel-based model search engine. Software & Systems Modeling, 11(2), 183–208. doi:10.1007/s10270-010-0167-7.

Madhavji, N. H., Fernández Ramil, J. C., & Perry, D. E. (2006). Software evolution and feedback: Theory and practice. Chichester: Wiley.

McQuillan, J. A., & Power, J. F. (2006). On the application of software metrics to UML models. In T. Kühne (Ed.), Proceedings of the 9th international conference on model driven engineering languages and systems, ACM, Lecture notes in computer science (Vol. 4364, pp. 217–226). doi:10.1007/978-3-540-69489-2_27.

Melnik, S. (2004). Generic model management: Concepts and algorithms, Lecture notes in computer science (Vol. 2967). Berlin and London: Springer.

Mens, T., & D’Hondt, T. (2000). Automating support for software evolution in UML. Automated Software Engineering, 7(1), 39–59. doi:10.1023/A:1008765200695.

Mens, T., Lucas, C., & Steyaert, P. (1998). Supporting disciplined reuse and evolution of UML models. In J. Bézivin, & P. A. Muller (Ed.), UML ’98: Beyond the notation, Lecture notes in computer science (Vol. 1618, pp. 378–392). Berlin and London: Springer. doi:10.1007/978-3-540-48480-6_29.

Mens, T., van der Straeten, R., & Simmonds, J. (2003). Maintaining consistency between UML models with description logic tools. In ECOOP workshop on object-oriented reengineering, Lecture notes in computer science (Vol. 3031).

Mens, T., & D’Hondt, T. (2000). Automating support for software evolution in UML. Automated Software Engineering, 7(1), 39–59. doi:10.1023/A:1008765200695.

Mohagheghi, P., & Dehlen, V. (2009). Existing model metrics and relations to model quality. In ICSE Workshop on Software Quality, 2009, WOSQ’09, IEEE, (pp. 39–45). Piscataway, NJ.

Moody, D. L. (2005). Theoretical and practical issues in evaluating the quality of conceptual models: Current state and future directions. Data & Knowledge Engineering, 55(3), 243–276. doi:10.1016/j.datak.2004.12.005.

Murta, L. G., Oliveira, H., Dantas, C., Lopes, L. G., & Werner, C. (2007). Odyssey-SCM: An integrated software configuration management infrastructure for UML models. Science of Computer Programming, 65(3), 249–274. doi:10.1016/j.scico.2006.05.011.

Ramírez, A., et al. (2004). ArgoUML user manual v0.16. http://argouml.tigris.org.

Ricci, F., Rokach, L., Shapiro, B., & Kantor, P. B. (Eds.). (2011). Recommender systems handbook. New York and London: Springer.
Roth, A. (2012). *Quality staged model evolution in Ecore Libraries*. Master Thesis, RWTH Aachen University, Aachen, Germany, https://www2.swc.rwth-aachen.de/?post_type=thesis&p=427.

Roth, A. (2013). A metrics mapping and sources. http://goo.gl/ruqFpi.

Roth, A., Ganser, A., Lichter, H., & Rumpe, B. (2013a). Proactive quality guidance for model evolution in model libraries. In *Models and evolution, proceedings of the workshop on models and evolution, colocated with ACM/IEEE 16th international conference on model driven engineering languages and systems* (MoDELS 2013), CEUR, CEUR workshop proceedings (Vol. 1090, pp. 50–59).

Roth, A., Ganser, A., Lichter, H., & Rumpe, B. (2013b). Staged evolution with quality gates for model libraries. In *Proceedings of the 1st international workshop on: (Document) changes: Modeling, detection, storage and visualization*, ACM, pp. 1–8.

Rumpe, B. (2012). *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring* (2nd ed.). Berlin: Xpert.press, Springer.

Segal, J. (2003). The nature of evidence in empirical software engineering. In *Eleventh annual international workshop on software technology and engineering practice*, IEEE, (pp. 40–47), doi: 10.1109/STEP.2003.33.

Stachowiak, H. (1973). *Allgemeine Modelltheorie*. S.l.: Vienna Springer.

Steinberg, D., Budinsky, F., Paternostro, M., & Merks, E. (2008). EMF—Eclipse Modeling Framework, 2nd edn. The Eclipse series, Addison Wesley, Upper Saddle River, NJ.

Struber, D., Rubin, J., Taentzer, G., & Chechik, M. (2014). Splitting models using information retrieval and model crawling techniques. In S. Gnesi, & A. Rensink (Ed.), *Fundamental approaches to software engineering*, Lecture notes in computer science (Vol. 8411, pp. 47–62). Berlin: Springer. doi: 10.1007/978-3-642-54804-8_4.

Sun, Y., White, J., & Gray, J. (2009). Model transformation by demonstration. In A. Schürr, & B. Selic (Ed.), *Model driven engineering languages and systems*, Lecture notes in computer science (Vol. 5795, pp. 712–726). Berlin: Springer. doi: 10.1007/978-3-642-04425-0_58.

Swanson, E. B. (1976). The dimensions of maintenance. In R. T. Yeh, & C. V. Ramamoorthy (Ed.), *ICSE ’76 proceedings of the 2nd international conference on software engineering*, IEEE, pp. 492–497.

Taylor, R. N., Gall, H., Medvidovic, N., Sun, Y., Gray, J., & White, J. (2011). MT-scribe: An end-user approach to automate software model evolution. In *Proceedings of the 33rd international conference on software engineering*, ACM, ICSE, pp. 980–982. doi: 10.1145/1985793.1985966.

Ujhelyi, Z., Bergmann, G., Hegedûs, Á., Horváth, Á., Izsó, B., Ráth, I., et al. (2015). EMF-IncQuery: An integrated development environment for live model queries. *Science of Computer Programming*, 98, 80–99. doi: 10.1016/j.scico.2014.01.004.

van Deursen, A., Visser, E., & Warner, J. (2007). Model-driven software evolution: A research agenda. Technical report series. www.st.ewi.tudelft.nl/~arie/papers/modse/modse2007.pdf. Accessed 5 Feb 2014.

van Enckevort, T. (2009). 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*, ACM, New York, NY.

Vclav Rajlich. (1997). A model for change propagation based on graph rewriting. In *Proceedings of the international conference on software maintenance (ICSM’97)*, pp. 84–91.

Voigt, K. (2011). *Structural graph-based metamodel matching*. PhD, TU Dresden, Dresden.

Wedemeijer, L. (2001). Defining metrics for conceptual schema evolution. In G. Goos, J. Hartmanis, J. Leeuwen, H. Balsters, B. Brock, & S. Conrad (Eds.), *Database schema evolution and meta-modeling*, Lecture notes in computer science (Vol. 2065, pp. 220–244). Berlin: Springer. doi: 10.1007/3-540-48196-6_13.

Weimer, M., Karatzoglou, A., & Bruch, M. (2009). Maximum margin matrix factorization for code recommendation. In L. Bergman, & A. Tuzhilin (Eds.), *RecSys ’09 proceedings of the third ACM conference on Recommender systems*, ACM (pp. 309–312). doi: 10.1145/1639714.1639775.

Wenzel, S., Hutter, H., & Keller, U. (2007). Tracing model elements. In *IEEE international conference on software maintenance (ICSM 2007)*, (pp. 104–113). doi: 10.1109/ICSM.2007.4362623.

Wüst, J. (2014). SDMetrics: The software design metrics tool for the UML. http://www.sdmetrics.com/.
Andreas Ganser is a Ph.D. student at research group Software Construction, RWTH Aachen University, Germany. He received his diploma degree in the domain of automated testing. He is now project lead for the Harvest, Evolve, and Reuse Models Easily and Seamlessly (HERMES) project. His scientific interests and contributions comprise several organized conferences and workshops and papers describing model libraries, mining, evolution, and recommenders.

Horst Lichter is a professor of computer science and head of the research group Software Construction at RWTH Aachen University, Germany. He received a diploma degree in computer science and economics from Technical University Kaiserslautern, Germany. He was at ETH Zurich and at the University of Stuttgart, where he obtained his Ph.D. Following this, he worked as project manager at the Union Bank of Switzerland, Zurich, and later as a scientist at ABB Corporate Research, Heidelberg, Germany. His research interests include software architecture, software metrics, and development processes.

Alexander Roth is a Ph.D. student in the Department of Software Engineering at the RWTH Aachen University, Germany. He received his M.Sc. degree in computer science from the RWTH Aachen University in the domain of model evolution in model libraries. His research interests include model evolution, code generator development, and generator product lines.
Bernhard Rumpe is chair of the Department for Software Engineering at the RWTH Aachen University, Germany. Prior to this, he chaired the Software Engineering Institute at TU Braunschweig. He obtained his Ph.D. and habilitation from TU Munich. His main interests are software development methods and techniques that benefit from both rigorous and practical approaches. These include the impact of new technologies, such as model-engineering based on UML-like notations, domain-specific languages and evolutionary, test-based methods, software architecture, as well as the methodical and technical implications of their use in industry. He has furthermore contributed to the communities of formal methods and UML. He is author and editor of eight books and Editor-in-Chief of the Springer International Journal on Software and Systems Modeling. He is co-founder and steering-committee member of the GI expert committee on “Modeling” in Germany, program-committee chair, PC member, workshop organizer, etc., at various organizations.