An empirical study of manual abstraction between class diagrams and code of open-source systems

Software and Systems Modeling (2025) 24:1797–1823
https://doi.org/10.1007/s10270-025-01289-y

SPECIAL SECTION PAPER

An empirical study of manual abstraction between class diagrams and code of open-source systems

Wenli Zhang[1] · Weixing Zhang[1] · Daniel Strüber[1,2] · Regina Hebig[3]

Received: 15 July 2024 / Revised: 2 December 2024 / Accepted: 19 March 2025 / Published online: 15 May 2025

Abstract

Models play a crucial role in software design, analysis, and supporting new maintainers. However, over time, the benefits of models can diminish as system implementations evolve without corresponding updates to the original models. Reverse engineering methods and tools can help maintain alignment between models and implementation code. Yet, automatically reverse-engineered models often lack abstraction and contain extensive details that hinder comprehension. Recent advancements in AI-based content generation suggest that we may soon see reverse engineering tools capable of human-grade abstraction. To guide the design and validation of such tools, we need a principled understanding of manual abstraction—a topic that has received limited attention in existing literature. In pursuit of this goal, our paper presents a multiple-case study of model-to-code differences, examining nine substantial open-source software projects obtained through repository mining. We manually matched source code from projects comprising 4983 classes, 26k attributes, and 54k operations to 523 model elements (including classes, attributes, operations, and relationships). These mappings precisely capture discrepancies between provided class diagram designs and actual implementation code. By analyzing these differences in detail, we derive a taxonomy of difference types and provide a well-organized list of cases corresponding to identified differences. Our findings have the potential to contribute to improved reverse engineering methods and tools, propose new mapping rules for model-to-code consistency checks, and offer guidelines to avoid over-abstraction and over-specification during the design process.

Keywords Software design · Modeling · Abstraction

1 Introduction

Models serve as essential tools for software analysis and design [1]. By capturing structural and behavioral aspects of systems at a suitable abstraction level, they aid in various critical tasks. First and foremost, models provide a blueprint for implementing software through both manual programming and automated code generation. This blueprint ensures that developers have a clear understanding of the system's intended structure and behavior. Moreover, models facilitate effective communication among stakeholders-developers, architects, and domain experts-by providing a visual representation that transcends technical jargon. When discussing system requirements, design decisions, or changes, referring to a model simplifies complex discussions.

This simplification is achieved by abstraction. Abstraction plays a core role in computational thinking. Wing describes the abstraction process as "deciding what details we need to highlight and what details we can ignore"[2] and Qian and

Communicated by G. Taentzer, A. Cicchetti, A. Pierantonio, and T. Kühne.

✉ Daniel Strüber
danstru@chalmers.se

Wenli Zhang
wenliz@student.chalmers.se

Weixing Zhang
weixing@chalmers.se

Regina Hebig
regina.hebig@uni-rostock.de

[1] Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Gothenburg, Sweden

[2] Department of Software Science, Radboud University, Nijmegen, The Netherlands

[3] Institute of Computer Science, University of Rostock, Rostock, Germany

Choi further conclude that such abstraction is crucial when thinking about complex problems [3]. In the context of software modeling, we refer to the omission and generalization of the information shown in the model compared to the represented system (in our case the source code). Thus, abstraction is a form of difference between source code and model.

However, maintaining consistency between models and the actual implementation code presents challenges [4]. During the development lifecycle, models evolve alongside code changes. Unfortunately, this evolution doesn't always happen seamlessly. Developers may inadvertently neglect to update the model after modifying the code, leading to discrepancies [5]. Additionally, intentional deviations from the model can occur during implementation due to practical constraints, design trade-offs, or unforeseen complexities [6, 7]. These inconsistencies can hinder system comprehension, making it harder for software engineers to navigate the codebase effectively.

To address this issue, reverse engineering methods and tools come into play [8–15]. These tools automatically extract models from existing code, bridging the gap between the abstract representation and the concrete implementation. Recent advancements in AI-based content generation, exemplified by large language models like GPT-4 [16], offer exciting possibilities. By mimicking human-grade abstraction, these models could enhance reverse engineering tools, producing more accurate and concise models directly from code. Imagine a tool that not only captures the structural elements but also abstracts away unnecessary details, resulting in models that strike the right balance between comprehensiveness and simplicity.

To our knowledge, no existing study has investigated manual abstraction by considering concrete model-to-code differences. Existing studies on manual abstraction are based on participant opinions and experiences [17], yet, do not study actual cases of models and code. Available consistency checking techniques are purely structural and do not take semantics of model elements into account [18–20]. However, semantics is key for understanding abstraction. Given that different systems have their own implementation structures, the desired functionalities require code structures that are interrelated while also accounting for the application of architectural and design patterns. These factors need to be considered jointly, which can only be achieved by careful manual studies of models and code.

In this paper, to close this gap, we present a multiple-case study on model-to-code differences. We investigate nine substantial cases with available models and implementation code, retrieved using the Lindholmen Dataset [21] and the SEART repository search engine [22].

To explore characteristics of model-to-code differences, we manually created mappings that precisely capture differences in real projects together with explanations of their origins. Our study focuses on class models, a particularly widely used model type, whose main diagram type are class diagrams. Class models are an especially important case for model-to-code consistency: as they model the domain of interest in terms of classes and relationships between them [23], they are intensively used in the early development stages to specify the system's structure. Maintainers benefit from using class models to understand the system's structure and then identify code locations to be modified [24], which requires the class model to remain an accurate representation of the system over time.

By pinpointing abstraction practices that are naturally applied by humans, our findings are valuable for tool developers in the reverse engineering and consistency checking domains, who aim for their tools to emulate human behavior.

In particular, we find that one cause of omission on class level are *superclass and subclass omission*; that is, inheritance structures are valuable for distinguishing what to include in and exclude from the model. For example, model elements connected to a superclass are more likely to be candidates for inclusion, whereas elements connected to a subclass are more likely to be omitted. We further identify a number of easy-to-apply best practices for abstraction, including *type parameter omission* and *relationship loosening*, acknowledging that collection types, types specified for attributes and operations, and particular association types, e.g., composition vs. aggregation, are often regarded as minor implementation details. Other omissions, e.g., of *parameter type* and *return type*, are particularly useful if the omitted information is already obvious through the name of the parameter or method at hand. Surprisingly, we find very few cases of *summary of elements*, e.g., by representing four source-code classes through one model-level class, suggesting that this practice appears to be less natural and needs less explicit support in tools.

Specifically, we make the following contributions:

- A taxonomy of model-to-code differences.
- A systematically elicited list of cases corresponding to the identified types of differences.
- A discussion of the potential uses of our taxonomy and case list.
- A replication package [25], which includes links to the models, code versions used, manually annotated models, reverse-engineered models, and corresponding comparison templates for the nine cases.

This paper is an extended version of our MODELS paper [26], which was based on a subset of five of the nine subject systems that we consider in the present paper. In the earlier paper, we laid out preliminary versions of our contributions but left it as future work to confirm them on unseen

cases. Based on a duality argumentation, we also speculated that an additional disagreement type—class pretense—might exist, which has not been observed in any of the five original projects. In this extended, paper, we now close this gap. Specifically, we make the following extensions: First, we conducted an additional phase of labeling and analysis on four newly selected projects, largely confirming our previous taxonomy while extending the list of cases and also observing that the speculated disagreement type of class pretense indeed exists. Methodology aspects and new results pertaining this second phase are documented the significantly revised Sect. 3 and 4, respectively. Second, we significantly extended our discussion of related work in Sect. 2. Third, we extended the discussion in Sect. 5, in particular, by substantially expanding on our contribution to theory building, but also addressing more directions to explain potential uses of our data, including the development of AI-based tools in MDE as well as teaching. Fourth, we make available an updated version of our dataset [25] including the data from the new labeling and analysis.

2 Related work

We now discuss related work in several directions: studies of abstraction, reverse engineering, consistency of models and code, and the use of models in software development.

2.1 Manual abstraction

Class diagrams used in software development can be overwhelming with volumes of information, making it challenging for software maintainers to understand system architecture [17]. To simplify class diagrams, understanding how software engineers manually create abstraction is crucial. For this purpose, Osman et al. surveyed developers to investigate how manual abstraction is created over code [17]. They found that developers considered it important to include the following elements in a class diagram: class relationships, meaningful class names, and class properties. Developers in the survey further claimed that *GUI-related information*, *Private and Protected operations*, *Helper classes*, *Library classes*, and *Interface classes* should be excluded from class diagrams [17]. Also, Baltes and Diehl's online survey found that most participants related sketches (including UML notations) to methods, classes, or packages but rarely to more detailed aspects, such as attributes [27]. What the above studies have in common is that they are based on participant opinions and experiences rather than studying actual models. As such, they are orthogonal to the present work, which is based on an artifact study.

In the context of computing education, Qian and Choi [3] propose a framework of abstraction in computational thinking together with a set of design guidelines for enhancing students' uptake of abstraction. Their framework includes three fundamental cognitive processes associated with abstraction—namely, *Filtering Irrelevant Information*, *Locating Fundamental Similarities*, and *Mapping out Problem Structures*—which they map to abstraction dimensions such as classification, generalization, and simplification. Since their framework is applicable to both structural as well as procedural concerns, and decoupled from specific representations, it is more abstract than ours. This leads to a different set of advantages, such as being broadly applicable in different contexts in computing education, and disadvantages, such as having fewer concrete insights how abstraction looks like for a particular representation.

Later in the paper, namely, in Sect. 5.1, we will explore how our findings related to specific insights from this line of work.

2.2 Reverse-engineering class diagrams

Müller et al. [28] defined reverse engineering in [29] as the process of analyzing a system to identify its components and their relationships and extract and create system abstractions and design information. Reverse engineering tools today can automatically generate class diagrams from the current code, even though they can only make limited abstraction decisions. One of the earliest approaches, PTIDEJ, was developed by Guéhéneuc et al. [24] and infers relationships in class diagrams. A well-known tool is MoDisco [8], which is a framework for model-driven reverse engineering that extracts information from existing artifacts to generate different representations of the system. Koschke [9] reviews techniques for *architecture reconstruction*, which refers to reverse engineering that allows concluding on the architecture of the system. He concludes that while it seems trivial to generate class diagrams from code, the challenge is in identifying what should be shown and what not. As reverse-engineered diagrams are often cluttered [10], approaches for abstraction by rules and using machine learning were developed. In the former group, Egyed [11, 12] defined semantic abstraction rules, such as a rule to substitute two relationships that form an indirect relationship with one direct relationship. Booshehri and Luksch [13] developed an approach that utilized semantic web techniques based on the V-OntModel. Another approach for Ontology-based model abstraction comes from Guizzardi et al. [30]. They implement a collection of abstraction rules, such as, similar to Egyed, abstractions by introducing direct relationships to substitute indirect relators. In the latter group, Osman et al. experimented with a supervised classification algorithm to condense class diagrams [31]. Thung et al. extended the work of Osman et al. by adding network metrics (e.g., closeness centrality) and achieved a 9% improvement [10]. Yang et al. [32] introduce MCCondenser which is a

tool that requires small amounts of labeled data to learn what aspects of a class diagram should be shown and what not. Compared to Thung et al. they achieve an improvement of 10-20%. What is common to all these tools and techniques is that there is still room for improvement when it comes to abstraction – especially with the recent progress in AI/ML. One precondition for that is a better understanding of what abstraction performed by humans actually looks like.

Common to these works is that they use hardcoded or statistically retrieved heuristics to emulate human abstraction, typically based on metrics. Our manual categorization of differences is complementary to such heuristics, and could inform the development of improved reverse engineering approaches that incorporate knowledge of what human abstraction looks like, as we further discuss in Sect. 5.4.

2.3 Consistency check(s) between code and design

Reverse-engineered diagrams are generated by extracting information from code in the absence of a design (typically represented as a model) that can be referenced. In contrast, consistency checks between code and design rely on the existence of a design. Antoniol et al. [18] argue for the need to maintain and reconstruct traceability, i.e., matches between design and code. Different models and methods exist to check consistency between code and design. Antoniol et al. [18] and Dennis et al. [19] find that class names play the most critical role in recovering mappings or traces between entities in design and constructs in code. However, they also find that a fourth of the class names in their dataset were different in code and model, making it difficult to match classes in code and model to each other. The reasons for such deviations between unmatched classes have not been systematically studied yet.

In addition, there is existing work on checking and avoiding inconsistencies between model design and code. Riedl-Ehrenleitner et al. [33] proposed a new approach to instantly check and notify developers of inconsistencies between models and source code during development, thereby effectively preventing the introduction of inconsistencies. Chavez et al. presented a model-based approach in [34] for testing the conformance of software implementations to their design models in Model Driven Engineering. They developed a prototypical tool to efficiently check for inconsistencies, which they successfully applied to the Eclipse UML2 projects, revealing inconsistencies between models and their implementations.

Jongeling et al. proposed an extension of the OpenMBEE platform in [35], where code is treated as a view within the platform and explicitly linked to the generated documentation and model. This aims to analyze the gaps between models and code, thereby improving consistency between design and implementation. In [36], they focused on the mechanism for automatically establishing traceability links between system model elements and parts of the code. This automated consistency checking mechanism aims to prevent errors during development and ensure consistency between models and code, thereby avoiding delays or erroneous implementations. Furthermore, in [37], they conducted a design science study, developed a consistency checking tool, and integrated it into an existing industrial system and software engineering environment. Their case study demonstrated that introducing lightweight consistency checks into the continuous integration pipeline is beneficial for consistency management, and they also discussed the non-technical challenges of introducing consistency checks in an industrial setting. Additional work in this direction is regularly presented at the workshop on *Software Architecture Erosion and Architectural Consistency (SAEroCon)*, which, as one of its themes, deals with the linking of high-level architectures with available implementation representations.

Besides the desire to understand abstraction, this line of work motivates the need to understand how code and models are typically different, to be able to provide tooling to reconstruct their relationship. Our paper is a contribution toward addressing this need for the case of class diagrams and implementations.

2.4 Models in software development practice

Baltes and Diehl conducted an online survey of 394 participants on the use of sketches and diagrams and found that most participants associated their sketches with methods, classes, or packages rather than with code at a lower level of abstraction [38]. Chung et al. questioned 230 contributors from 40 different FOSS projects and interviewed eight of them about the use of sketches and diagrams. They found that the use of sketches and diagrams in FOSS was not frequent. This finding adds to the motivation of our study: it makes us confident that the manual abstraction characteristics we provide would contribute to enabling software developers to understand how to create class diagrams at an appropriate level of abstraction to fit the implementation structure of the required system and will play a key role in helping software engineers understand and avoid misunderstanding the implementation structure of the system.

It is difficult to access industrial projects that contain class diagrams and their corresponding source code. Therefore, we decided to leverage models from FOSS. Karasneh et al. [39] used a crawling approach to obtain so far more than 700 UML class diagrams, and then converted them into xmi and stored them in a searchable database. Hebig et al. collected more class diagrams and then created the Lindholmen dataset [21] which provides model context in addition to model artifacts. The dataset was obtained using a semi-automated approach in which they collected UML stored in images (.jpeg, .png, .gif, .svg, and .bmp) and standard formats (.xmi and .uml files)

by randomly scanning 10% of all GitHub projects (1.24 million). As a result, they obtained a list of 3,295 open-source projects, including 21,316 UML models (later extended to 24,000 projects with together 93,000 UML models). The Lindholmen dataset is the first corpus established in the modeling community. As such, this earlier paper is fundamental for the present study, as it enabled the identification of seven of our nine cases.

3 Methodology

We performed a multiple-case study [40] in this paper, aiming to answer the following questions.

RQ1: Which types of differences can be found between models and corresponding source code?

RQ2: How can we classify forms of abstraction between model and source code?

RQ3: How can we classify forms of non-abstraction difference between model and source code?

We studied these questions in two separate iterations, each with the steps of *project selection*, *selection of model and source code versions*, and *analysis*. The first iteration, which was presented in the earlier conference version of this paper [26], was based on an initial selection of five projects. In the second iteration, we considered four additional projects. We now present the details for the steps, addressing differences of the two iterations.

3.1 Project selection

In the first iteration, we retrieved projects using the Lindholmen dataset [21], a collection of open-source projects from GitHub that include UML models. The collected models are stored in image formats (.jpeg, .png, .gif, .svg, and .bmp) and standard formats (.xmi and .uml files). The dataset lists over 24,000 open-source projects which together include 93,000 UML files [21, 41]. The advantage of using this dataset as a starting point is that it provides us with access to cases that include code as well as models.

In the second iteration, we used the same dataset again, but eventually observed a scarcity of large, comprehensive projects that we could use to further extend our findings. Therefore, as an additional data source we used a list of GitHub repositories that contained PlantUML files, obtained from a dataset of Romeo et al. [42] who created it using the SEART repository search engine [22]. PlantUML is a notation that supports the automated creation of a visual UML diagram from plain text. In contrast to UML diagrams stored in generic image formats, PlantUML files can be easily identified in a large collection of available project files, using their file ending (.puml, .plantuml), which was our motivation for directing this project selection effort at them. To ensure the identification of large, widely used projects, the list was obtained by searching for repositories with at least 10 developers and at least 100 stars.

The analysis of the source code and models is very time-consuming, due to the need to understand the semantics of the system for making correct judgments. On the other hand, we do not expect to be able to fully understand typical differences by studying only a single project, so we needed to study multiple projects. In the first iteration, we set out to study five projects. In the second iteration we added four additional projects.

To identify suitable projects within the dataset, we defined the following criteria: First, the projects should include both, class diagrams and the source code of the modeled system. Second, the models should be available in an image format. The reason for that is that models in .xmi or .uml format requires extra effort and—often—specific tools to be opened. Selecting class diagrams in image format can save us time and allow us to acquire information on the class diagrams directly. For the second iteration, we loosened that requirement to also allow for PlantUML files as these can easily be converted into graphical models for the analysis. Third, the model must be created manually and not with the help of a reverse engineering tool. Note that it was not trivial to automatically exclude models that are the result of reverse engineering. Therefore this had to be done manually, which took a lot of time. For pragmatic reasons, we ended up studying Java projects, as it was easiest to identify suitable project candidates in that language.

Iteration 1. We iterated through the Lindholmen dataset checking projects for these criteria. To deal with the abundance of available repositories in the Lindholmen dataset, we applied a two-stage selection strategy: 1. From the full list of all UML files, we considered entries both randomly and one-by-one from the top of the list. This strategy led to the identification of three projects (EAPLI_PL_2NB, RaiseMeUp, ZooTypers) that satisfy the selection criteria. 2. Based on the experience from the first stage, given the low success rate of the approach taken in it, we narrowed down our preselection, by using an available list with class diagrams from the Lindholmen dataset for which an image was available [43]. This list contained 415 class diagrams identified as forward diagrams, whose associated projects we considered one after another until a target number of five projects was reached. This led to the identification of two further projects (FreeDaysIntern, NeurophChanges). The search was terminated when the five projects have been identified.

Iteration 2. During the project selection in iteration 2, we benefit from the insights made during iteration 1. Specifically, since the first stage in iteration 1 led to a low success rate, we only performed the second stage from iteration 1, using an available narrowed-down list of projects with class diagrams available as images. This led to the identification of the other

2 projects (PatrickFromTheMOOC, myblog) that satisfied the selection criteria. Considering the list of GitHub repositories that contained PlantUML files (described above), we applied the same criteria as for the Lindholmen dataset. We iterated through that list to identify projects that meet the criteria, until we reached a predefined number of 9 repositories in total. This led to the identification of the 2 additional projects (Orekit and Artemis).

Overall list of projects. Table 1[1] summarizes the basic information of these nine projects, including their domains, active periods, etc., while Table 2 summarizes their status regarding files, source code, and model elements. While only two of our considered projects are still actively developed, our projects spanned a range of active project duration, between 1 and 258 months of activity. ZooTypers is an android project of an animal-themed typing game. RaiseMeUp is a GUI project for keeping electronic pets, e.g., fish. EAPLI_PL_2NB is a web application for recording transactions, e.g., income and expense. FreeDaysIntern is a web application used for creating labor billing time sheets and finally, NeurophChanges is a lightweight neural network framework to develop common neural network architectures. PatrickFromTheMOOC is a GUI project for reviewing books and films. myblog is a web application for managing blogs. Orekit is a low-level space dynamics library, offering features for describing and handling space elements. Artemis is a platform for interactive learning with features such as programming exercises, quizzes, modeling tasks, and individual feedback.

3.2 Selection of source code and model versions

For each project, we selected one class diagram to study. In cases where only one class diagram was included, we selected that class diagram for the study. In case we found a class diagram that was updated over time, we decided to select the latest version of the model. Finally, if a project included multiple class diagrams, e.g., presenting different system parts, we selected one of them randomly for the study. This was the case for projects 3 and 8, where multiple class diagrams were used to present different features in the system. Note that we assume here that the class diagram shows the complete model. The selection of the version of the source code to study was more complex. All of the methodological steps in this process were completed manually. Table 3 presents the result of this process: Repository versions identified for the nine projects, in particular, versions for the source code and the model, which can, but do not need to be the same version.

We now describe the process and its steps in detail.

[1] The original project 2 is not accessible anymore. We link a fork of the project here that we made for the study.

3.2.1 Selection aim

Selecting a version of the source code that is too old, might lead to an overestimation of differences between the source code and the model because modeled elements might not yet be implemented. Note that this can also hold for the source code that is present when the model is committed, as the class diagram might be prescriptive and, thus, the development of the corresponding source code still has to follow. On the other hand, a descriptive model might in rare cases also be most similar to a slightly older version of the code. On the other hand, selecting a version of the source code that is too young, can also lead to an overestimation of differences, as the source code might have evolved. Thus, we would capture differences that are due to code evolution and it would be difficult to distinguish which differences are due to abstraction.

3.2.2 Selection process

Due to the reasons above, we aim to find the version of the source code that implements the highest number of classes, attributes, and operations shown in the model while minimizing the amount of additional code. However, given that most of the studied projects have hundreds of versions of source code, this assessment is not feasible.

Therefore, we worked with a heuristic, following the procedure shown in Fig. 1. We start with the assumption that the code version to select is likely to be close to the commit with which the selected model was committed. So we selected that code version as the starting point for the procedure. In step 1, we performed a high-level analysis of the mapping between that version of the source code and the model, focusing on the concepts modeled as classes, attributes, and operations in the diagram. In step 2, we performed the same mapping for the source code in the versions before and after that commit and identified the version that provided the best match, in terms of agreeing in most elements. Based on the outcome of this comparison, we would select the best match among the three considered code versions. If the previous or following code version lead to the best match, we would continue with step 3, considering the mapping between that version and the model, which leads to an iterative loop of moving backward or forwards in the revision history until we found a best match. The loop stops when we find no more improvement. In that case, we have reached step 4, returning the version of the source code that, among the studied ones, offered the best match to the model.

As a result, we work with the assumption that a) we have minimized the risk of overestimating the differences between the model and source code, and b) the source code does not include differences that are due to code evolution.

The process for selecting of model and source code versions was the same in iterations 1 and 2.

Table 1 The studied projects from iteration 1 (#1-#5) and iteration 2 (#6-#9) with meta-information: active period = time between first and last commit in months; #versions = the number of code versions found in the repository (counting each commit as one version); #contributors = the number of contributors to the repository; Note that the projects marked with * are still active and the active period was counted from the start of the project to Nov. 11, 2024.

ID	Name	Domain	Active period	#versions	#contributors
1	ZooTypers [44]	Android	~1.5	745	5
2	RaiseMeUp [45]	GUI	~1.5	24	2
3	EAPLI_PL_2NB [46]	Web App	~2	483	9
4	FreeDaysIntern [47]	Web App	~19	449	3
5	NeurophChanges [48]	GUI	~28	534	7
6	PatrickFromTheMOOC [49]	GUI	~1	67	1
7	myblog [50]	Web App	~3.5	18	2
8	Orekit [51]	Library	~258*	9699	57
9	Artemis [52]	Web App	~99*	9110	223

Table 2 The number of files, source code, and model elements for the studied projects in iteration 1 (#1-#5) and iteration 2 (#6-#9)

ID	Name	Files	source code			model elements		
			Classes	Operations	Attributes	Classes	Operations	Attributes
1	ZooTypers [44]	87	15	77	52	6	31	16
2	RaiseMeUp [45]	168	40	545	474	17	59	43
3	EAPLI_PL_2NB [46]	103	60	255	51	8	30	9
4	FreeDaysIntern [47]	302	92	502	216	8	35	15
5	NeurophChanges [48]	2270	259	1255	559	10	0	0
6	PatrickFromTheMOOC [49]	30	43	141	86	7	47	16
7	myblog [50]	171	45	438	211	8	0	16
8	Orekit [51]	7661	2731	29,479	14,052	7	18	4
9	Artemis [52]	10,205	1698	21,523	10,396	8	13	10

The latter two include the number of classes, operations, and attributes in source code and model

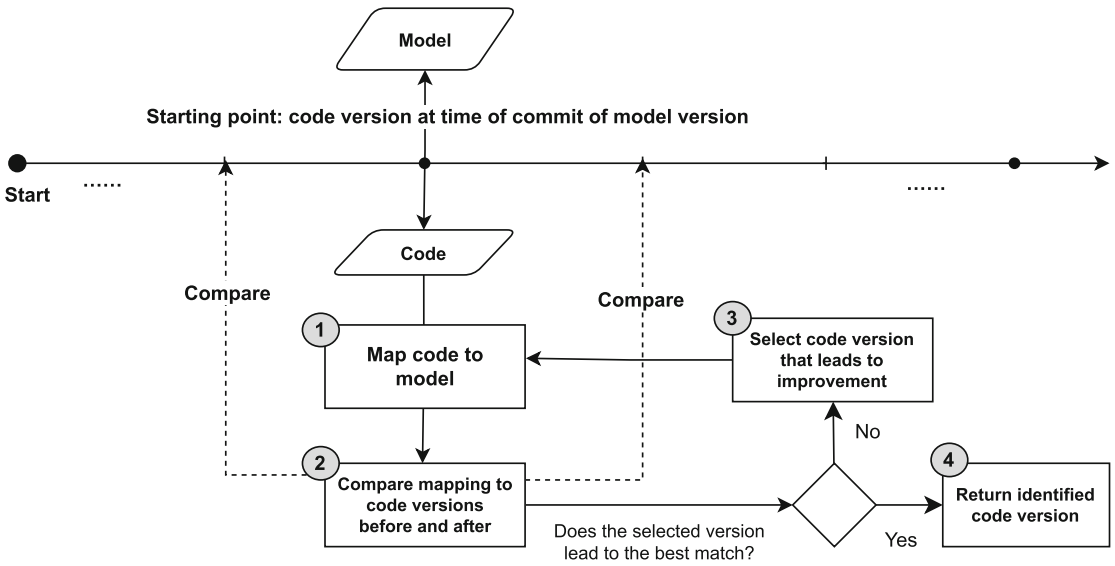


Fig. 1 Selection of the considered source code version

Table 3 Repository versions used in iteration 1 (#1-#5) and iteration 2 (#6-#9)

ID	Name	Source code	Model
1	ZooTypers [44]	https://github.com/ZooTypers/ZooTypers/tree/7591f15	https://github.com/ZooTypers/ZooTypers/tree/7591f15
2	RaiseMeUp [45]	https://github.com/WenliZhang1102/RaiseMeUp/tree/f3796c5	https://github.com/WenliZhang1102/RaiseMeUp/tree/d32e2f6
3	EAPLI_PL_2NB [46]	https://github.com/AntonioRochaOliveira/EAPLI_PL_2NB/tree/facb2e8	https://github.com/AntonioRochaOliveira/EAPLI_PL_2NB/tree/35d280a
4	FreeDaysIntern [47]	https://github.com/fmacicasan/FreeDaysIntern/tree/b306738	https://github.com/fmacicasan/FreeDaysIntern/tree/b306738
5	NeurophChanges [48]	https://github.com/tekosds/NeurophChanges/tree/848a3aa	https://github.com/tekosds/NeurophChanges/tree/848a3aa
6	PatrickFromTheMOOC [49]	https://github.com/PatrickFromThe29/PatrickFromTheMOOC/tree/5846aa2	https://github.com/PatrickFromThe29/PatrickFromTheMOOC/tree/e2b99c7
7	myblog [50]	https://github.com/jdkcn/myblog/tree/6494eaf	https://github.com/jdkcn/myblog/tree/6494eaf
8	Orekit [51]	https://github.com/CS-SI/Orekit/commit/4314309	https://github.com/CS-SI/Orekit/commit/4314309
9	Artemis [52]	https://github.com/ls1intum/Artemis/commit/8885e22	https://github.com/ls1intum/Artemis/commit/8885e22

3.3 Analysis

Our analysis consisted of detecting the differences between model and code and categorizing these differences, and was done entirely manually. The analysis is documented in our online appendix [25]..[2]

3.3.1 Difference detection

For detecting the differences between the model and source code, we initially hoped to use an automatic reverse engineering tool to help us visualize the code and ease the comparison. However, after exploring different tools (EA and IntelliJ IDEA) we had to observe that this did not work very well as relationships might be represented differently or incomplete and as it was necessary to understand the semantics of the code, which required us to read the code. In consequence, we worked directly with the source code during the analysis. We used IntelliJ IDEA to generate illustrating images of some of the observed differences so that they can be illustrated in this paper.

For the analysis, we first mapped modeled classes to source code files. We then manually create one-to-one mappings from the model elements to the source code constructs in terms of attributes and operations. For any suspected non-conformance between model and code, in terms of attributes and operations, we studied the source code in detail to fully understand the roles of the related classes, functionalities of attributes, and operations associated with these classes. For studying the differences between the relationships, we took attributes, operations, and inheritance into account. It was again necessary to conduct a complete manual analysis, considering roles of classes and semantics of modeled relationships.

In Sect. 4, we present various examples of different model-to-code mappings obtained in this analysis.

3.3.2 Assessed characteristics

To ensure a systematic comparison we worked with a template, which included space for information about the project, the model, the source code, the differences between model and code, and additional notes. For the project we captured meta-information. For the model, we assessed the commit identifier of the commit that added the model, the creation date of the model as well as the number of classes, attributes, operations, and relationships shown. For the code, we assessed the commit identifier of the commit that led to

[2] Filled-out comparison templates for all projects can be found the directory *Comparison Templates for Nine Projects* As an example, consider the template for the project Orekit: https://osf.io/xz9n4?view_only=34b153d3b7704f4a84befd661dc9c6a1.

the studied version of source code, creation date, number of classes, attributes, operations, and relationships as well as whether the elements in the model were covered in the code.

3.3.3 Classification

Finally, to answer the research questions stated at the beginning of this section, we used the raw data collected in the previous steps to create a classification. We inspected the differences line-by-line and derived *cases* such as *attribute omission* or *relationship summary* that indicate a recurring type of difference, until all of the differences were covered by a case. The list of these cases collectively answers RQ1. We further noted that not all differences are related to actual abstraction, as some of them either represent a *disagreement* that cannot be attributed to abstraction or an *inaccuracy*. Assigning our cases to these three main categories of differences led to the answers for RQ2 (abstraction) and RQ3 (disagreements and inaccuracies). This step was conducted after both iterations and led to the identification of new cases in the newly considered projects from iteration 2.

The identified inaccuracies help shed a better light on why it can be difficult to automatically create traces from source code to models (see the discussion in Sect. 2.3.

We present these categories and cases in detail, including the projects in which they occurred and the number of occurrences, in Sect. 4.

3.4 Reliability measures

The methodological steps described in this section were generally performed by the first author of this paper, with input and feedback from the other authors. In particular, the first and fourth authors together developed the template for difference analysis by considering specific cases and developed the first version of the classification, which was refined with input from all authors. The first author also identified cases that required discussion and brought them forward to the group of authors. All critical cases were discussed and eventually resolved with consensus from all authors. In addition, making available our detailed dataset [25] contributes to the reliability and reproducibility of our study.

3.5 Time consumption

Both the selection of the right code version and the analysis were very time-intensive. In general, the fewer attributes and operations from the model covered by the source code, the more time was required to check other attributes and operations in the source code to exclude a match during the data selection phase. In iteration 1, for three of the five projects studied, the final selected version of the source code is the one associated with the commit of the model. In these cases, it took around half a day to select the version of source code to study per project. For the other two projects in iteration 1, the time spans between the commit of the model and the commit of the selected version of the source code were both less than ten days. In both cases, the effort to identify the source code to study was about one full day.

In iteration 2, for one of the four projects, the time span between the commit of the model and the commit of the selected version of the source code was less than ten days. In this case, the identification of the source code version to analyze took around half a day. For the other three projects, the selected version of the source code is the one associated with the commit of the model. For one of them, it took only around one hour to identify the correct code version. This was possible because there are altogether less than 20 versions of the source code in the repository. For the other two projects, the same task took around half a day each. These two projects include over nine thousand versions of the source code each. However, in one of them, versions of the source code that clearly cover everything shown in the model are quite close in time to the commit of the model (and to the selected version of the source code). Since we could easily exclude these versions of the source code we could reduce the time needed for the selection. For the second of these two bigger projects, we identified quickly that hundreds of versions of the source code have the same coverage of model elements. Thus, we discovered quickly that the model was committed together with the implementing code.

The analysis took even more time. In general, the bigger the source code, the longer it took to analyze the system. For example, checking whether multiple classes/attributes/operations are represented by a given model element requires a comprehension of the complete source code. In iteration 1, for three of five projects analyzed, the number of model elements is lower than seventy. In this case, it took around a day per project of the three projects. For another project, the model has over 140 model elements. In this case, the analysis took around three days. This is mainly because the presence of more model elements led to the existence of more relationships between the classes. Given the higher number of classes in this project compared with other projects, we especially needed to identify direct relationships between the two classes and indirect relationships between two classes via a third class. This is because the IDE, e.g., VS Code we used did not support identifying the indirect relationships/references between the two classes in the code and vice versa. Thus, in order to identify the indirect relationships/references between the two classes we needed to read through the corresponding source code and identify the indirect relationships. This took more time than identifying the direct relationships between the two classes. In the smallest project with less than 20 model elements, the analysis was easier. We were able to jointly analyze classes and relation-

ships in a straightforward manner. In this case, the analysis took around half a day.

In iteration 2, one of the four projects analyzed, the model had around 80 elements. In this case, the analysis took around half a day. This is a similar time as it took for three projects in iteration 1, that had between 50–70 model elements. Still, the analysis took no more time as fewer differences between model and code were discovered compared to these 3 projects from iteration 1. Thus, in this case, it took half a day less than those three cases. For the other three projects analyzed in iteration 2, the number of model elements ranged from 34 to 37, also these cases took around half a day each. Even though these three projects have half the model elements as the project above, new cases of abstraction were found in all of them. Classifying those previously unseen cases took additional time.

In total, for iteration 1, we studied 466 source code classes together with 1352 attributes and 2634 operations, which had to be matched to 49 classes together with 155 attributes and 83 operations and relationships from the models. For iteration 2, we studied 4517 source code classes together with 24,745 attributes and 51,581 operations from the source code, which had to be matched to 30 classes together with 46 attributes and 78 operations and relationships from the model.

4 Results

The first observation we made when inspecting the differences between source code and model was that they did not always lead to the models being a more abstract version of the source code. Not all differences can be considered abstractions in the classical understanding. Thus, we distinguish three types of changes and define them as follows:

Real abstraction are cases where the model uses elements that specify more general semantics or contain fewer details than what can be found in the source code.

Disagreement are cases where the model uses elements that specify more specific semantics or contain more or different details than what can be found in the source code.

Inaccuracies are differences that cannot be classified as a difference in level of specificity and detail, but rather as non-conceptual differences in representation.

Note that we do not distinguish in these definitions whether the model was created before or after the source code.

> **Observation:** Not all identified differences serve the purpose to create a more abstract (and thus, readable) model.

4.1 Real abstraction

We now report on the observed cases of real abstraction, that is, having elements in the model that specify more general semantics or contain fewer details than what can be found in the code. Table 4 summarizes these cases, grouped by the context to which they pertain: subsystems, classes, attributes, operations, and relations. For each of these categories, we report between one and six *cases*.

4.1.1 Omissions

The first and, probably, least surprising group of abstractions are the omissions. We find omissions of subsystems, superclasses, subclasses, classes, attributes, attribute types, types parameters for either attributes or operations, default values, operations, parameters, parameter names, return types, and relationships.

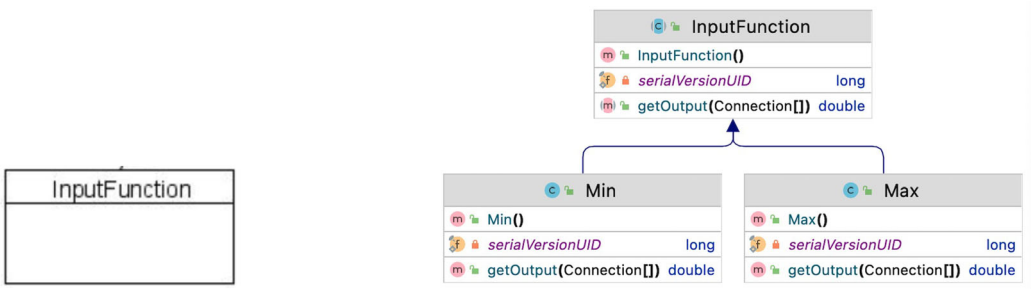
Subsystem omission is a special case, as not all models aim to show an abstraction of the complete system. We found subsystem omission in 7 out of the 9 cases (cases 3, 4, 5, 6, 7, 8 and 9). In six cases the model focused only on a very specific part and feature of the system and omitted all other system parts. In one case the model focused on the majority of the system and excluded a few specific features (e.g., Adapters and Image Recognition).

On the other hand, **class omission** affects classes (and their attributes, operations, and relationships) that are part of the system part illustrated by the model. We observed different reasons for that. In project 2, most classes related to the view of the MVC pattern were omitted while in projects 1, 5, 6 and 7 the models omitted all classes that were not important to understand the domain of the system, e.g., classes responsible for running frameworks, utility classes, and test classes were omitted. **Superclass omission and subclass omission**can be seen as two special cases of **class omission**. However, here the omitted classes are still represented in the model through their superclass or subclass, respectively. This representation is missing for most classes affected by **class omission**. For example, a case of **subclass omission** can be seen in Fig. 2a, b. Here the subclasses *Max* and *Min* derived from the superclass *InputFunction* were hidden in the model. Similarly, the figures also show a case of **operation omission** as the operations of *InputFunction* are not shown either in the model. Figure 3a, b shows an example of a case of **superclass omission.** The superclass *AbstractMatricesHarvester* inherited by the subclass of *DSSTHarvester* from the code was hidden in the model.

Another type of omission is the **type parameter omission.** This case occurred in attribute type, parameter type, and return type. Figure 4a shows an example of a *Set* < > attribute type of the attribute *accounts* in class *User* of project 7, but not what type of objects can be stored in the collection. In

Table 4 Cases of real abstraction

	Real Abstraction
Subsystems	
	Subsystem omission: The model focuses on one or more subsystems of the system only and omits all information about other parts of the code
Classes	
	Superclass omission: The superclasses in the inheritance structures in the source code are not shown in the model.
	Subclass omission: The subclasses in the inheritance structures in the source code are not shown in the model.
	Class omission: Classes present in the source code of the modeled system part are not shown in the model
	Class summary: Two or more classes present in the source code are shown as one class in the model
Attributes	
	Attribute omission: Attributes in the source code are not shown in the model
	Attribute summary: Multiple attributes in the source code are shown as one attribute in the model
	Attribute type omission: The type of an attribute in the source code is not shown in the model
	Attribute type generalization: An inner class of a class in the source code is omitted in the model and simultaneously, the type that inner class implements in the source code is shown as a more general type of an attribute in the model.
	Default value omission: An attribute in the source code has a default value that is not shown in the model
Operations	
	Operation omission: Operations in the source code are not shown in the model.
	Operation summary: Multiple operations in the source code are shown as one operation in the model
	Parameter omission: Parameters in the source code are not shown in the model
	Parameter name omission: Parameter names in the source code are not shown in the model
	Return type omission: The return type of a method in the source code is not shown in the model
	Type parameter omission: The types of objects that can be stored in collections as specified in the source code are not shown in the model, which only shows the type of the collection, or only the types of objects are shown, but not the information that there is a collection of these objects, or the types specified for either attributes or operations in the source code are omitted in the model.
Relationships (between classifiers)	
	Relationship omission: Relationships in the source code are not shown in the model
	Relationship loosening: An attribute (i.e., owned element) in the source code is modeled as a named association in the model (and not as a composition or aggregation)
	Relationship summary: For two classes that access each others' values indirectly via a third class in the source code a direct association is shown in the model



(a) Extract from the model of project 5.

(b) Visualization of corresponding code part from project 5.

Fig. 2 The subclasses and operations are hidden in the model

(a) Extract from the model of project 8.

(b) Visualization of corresponding code part from project 8.

Fig. 3 The superclass *AbstractMatricesHarvester* with its attributes and operations from the code are hidden in the model

the source code shown in Fig. 4b, the objects of *Account* are added as *Set<Account>*.

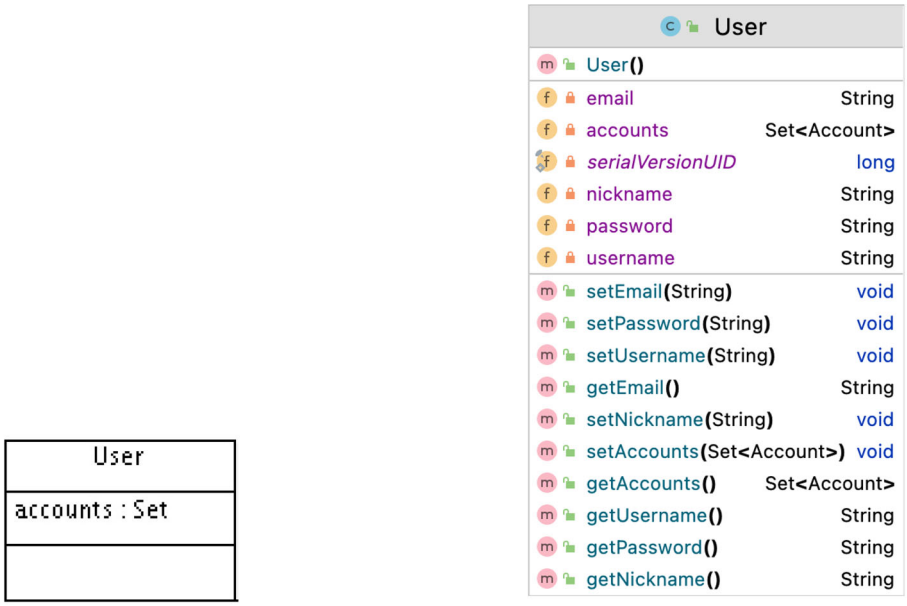
In another example, the type parameter omission happened for a parameter type. The example concerns the method *processMeasurements(observedMeasurements: List <ObservedMeasurement<?> >)*: DSSTPropagator (in class *SemiAnalyticalKalmanEstimator* of project 8). It was represented as *processMeasurements(measurements: List <ObservedMeasurement>)*: DSSTPropagator in the model, leaving out the generic parameter type <?>.

For the type parameter omission in the return type, in some cases, the model specifies that the type is a collection, e.g., a *Map* as for the return type of operation *listUser()* in class *DAO* of project 2, but not what type of objects can be stored in the collection. In the example case, these would be objects of type *Integer* and *User*. In other cases, the model specifies the type of the stored objects but omits the information that a collection is used.

4.1.2 Summaries

Another group of changes that we expected to find are cases where multiple elements from the source code are summarized. Indeed, we could find such cases, concerning classes, relationships, attributes, and operations.

For example, a case of **relationship summary** can be found in project 2. The naming of the classes in Fig. 5a indicates an *observer* design pattern [53], based on the naming of the class *PetObserver*, its operation *update(): void* of *PetObserver*, and the association between *PetObserver* and the



(a) Extract from the model of project 7. (b) Visualization of corresponding code part from project 7.

Fig. 4 Type parameter omission: the collection of *Account* objects from the source code is hidden in the model

observable class *Pet*. In the source code, the relationship is reverse and indirect. This is possibly due to the particular MVC architectural pattern applied in the source code. To be specific, *PetObserver* and *Pet* are Model classes, which are managed by the Controller class *RaiseMeUp*. Figure 5b illustrates that the operation *getCurrentPet()* of *RaiseMeUp* is invoked within *PetObserver* and the operation *getEnergy()* of *Pet* is further invoked with the help of *RaiseMeUp*.

4.1.3 Other abstractions

We observed two special cases of real abstraction.

The first case is **relationships loosening**, where an attribute, i.e., an owned element, in the source code is not modeled as a composition, but as a more general association. Figure 6a illustrates that in the model of project 6, a unidirectional association is modeled between the classes *Review* to *EvaluationReview*. In the source code (Fig. 6b) the type of the attribute *evaluationsReview* (*LinkedList<EvaluationReview>*) indicates that a group of instances of the class *EvaluationReview* is referenced by the class *Review*, representing a unidirectional relationship. These instances are further initiated within that class (in operation *Review(member: Member, note: float, commentaire: string)*).

The second case is **attribute type generalization**; for example, Fig. 7a, b shows that an inner class *ProcessingBuildJobItemListener* (in the class *LocalCIQueueWebsocketService* of project 9) from the source code is omitted in the model (an instance of class omission). Simultaneously, an attribute that has the type of the inner class is instead shown in the model with the more general type *EntryListener<LocalCIBuildJobQueueItem>*(that the inner class implements): *processingJobsListener: EntryListener<LocalCIBuildJobQueueItem>* and the attribute name *processingJobsListener* in the model is represented by that name of the inner class ProcessingBuildJobItemListener from the code.

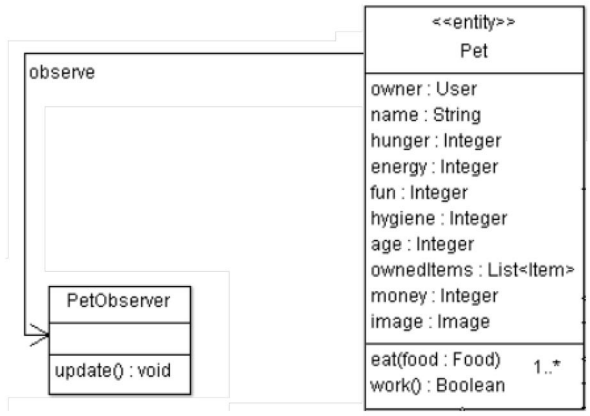
> **Observation:** Real abstraction cases observed include omissions, summaries, relationship loosening, and attribute type generalization.

4.2 Disagreements

We now report on the observed cases of disagreements, that is, having model elements that specify more specific semantics or contain more or different details than what can be found in the source code. Table 5 summarizes these cases, grouped by the context to which they pertain: classes, attributes, operations, and relations. For each of these categories, we report between one and six *cases*.

4.2.1 Pretenses

The most common form of disagreement that we observed is the pretense, where the model shows structures that are not present in the source code. We observed such pretense



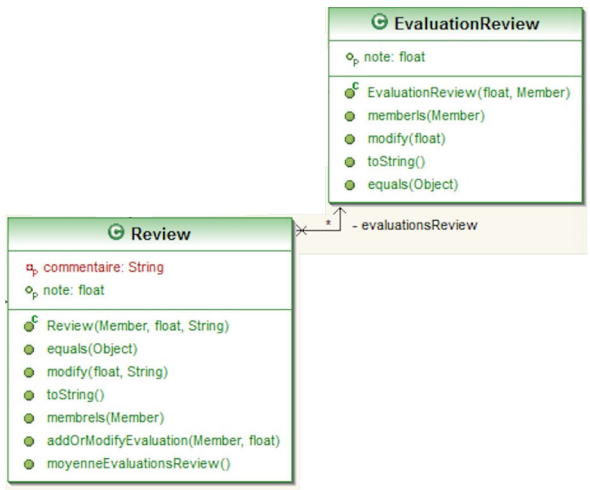
(a) Extract from the model of project 2 (re-layouted for readability).

```
public class PetObserver {
   // Complete implementation not shown here.
   public PetObserver() {
      /**
       * First, getCurrentPet() of RaiseMeUp invoked
       * by PetObserver.
       * Second, getEnergy() of Pet invoked by RaiseMeUp.
       */
      prevEnergy = RaiseMeUp.getCurrentPet().getEnergy();
   }
   // Complete implementation not shown here.
}

public class Pet {
   // Complete implementation not shown here.
   public int getEnergy() {
       return energy;
   }
   // Complete implementation not shown here.
}
```

(b) Visualization of corresponding code part from project 2.

Fig. 5 The model shows an association between *Pet* (origin) and *PetObserver* (target). In the source code *PetObserver* only indirectly accesses *Pet* via another class *RaiseMeUp*



(a) Extract from the model of project 4 (re-layouted for readability).

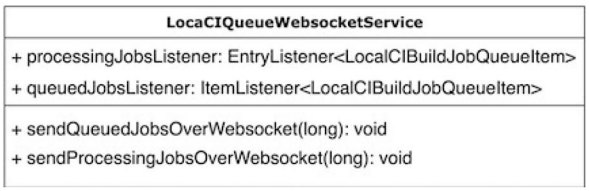
```
public class Review {
    // Complete implementation not shown here.
    private LinkedList<EvaluationReview>
            evaluationsReview = null;
    public Review(Member member, float note, String
            commentaire) throws BadEntry
    {
        // Complete implementation not shown here.
        this.evaluationsReview = new
            LinkedList<EvaluationReview>();
    }
    // Complete implementation not shown here.
}
```

(b) Visualization of corresponding code part from project 6 (re-layouted for readability).

Fig. 6 Relationship loosening: The ownership between *EvaluationReview* and *Review* from the source code is shown as a simple association in the model

occurrences for subclasses, attributes, operations, parameters, classes and relationships. Figure 8a, b shows an example of a class pretense from project 7. The class *Link* that is shown in the model is present as an attribute *urls: List<String>* of class *Blog* in the source code.

Figure 9a, b shows examples of an **subclass pretense**from project 2. The four subclasses *Dog*, *Cat*, *Fish*, and *Penguin* that are shown in the model are not present in the source code. There, only the superclass *Pet* can be found, which implements the difference between the pet types using an attribute *type*. Future work can study whether the superclasses in inheritance structures from the model are removed in the code. In the same figures we also see an **operation**



(a) Extract from the model of project 9.

```
public class LocalCIQueueWebsocketService {
   // Complete implementation not shown here.
   private final HazelcastInstance hazelcastInstance;
   // Complete implementation not shown here.
   public void init() {
       // Complete implementation not shown here.
       IMap<Long, LocalCIBuildJobQueueItem> processingJobs =
            hazelcastInstance.getMap("processingJobs");
       // Complete implementation not shown here.
       processingJobs.addEntryListener(new
            ProcessingBuildJobItemListener(), true);
       // Complete implementation not shown here.
   }
   // Complete implementation not shown here.
   private class ProcessingBuildJobItemListener
   implements EntryAddedListener<Long, LocalCIBuildJobQueueItem>,
      EntryRemovedListener<Long, LocalCIBuildJobQueueItem> {
       @Override
       public void entryAdded(com.hazelcast.core.EntryEvent<Long,
       LocalCIBuildJobQueueItem> event) {
          // Complete implementation not shown here.
       }
       @Override
       public void entryRemoved(com.hazelcast.core.EntryEvent<Long,
      LocalCIBuildJobQueueItem> event) {
          // Complete implementation not shown here.}}}
```

(b) Visualization of corresponding code part from project 9 (re-layouted for readability).

Fig. 7 Attribute type generalization: The inner class *ProcessingBuildJobItemListener* in the class *LocalCIQueueWebsocketService* from the source code is omitted in the model (a case of *class omission*). In the code a local variable *processingJobs* exists (in method *init()*) that has the collection type *IMap<Long, LocalCIBuildJobQueueItem>* and is filled with instances of the inner class *ProcessingBuildJobItemListener*. This is possible since the inner class implements the type *EntryAddedListener<Long, LocalCIBuildJobQueueItem>*. However, in the model instead of the local variable of type *IMap<Long, LocalCIBuildJobQUeueItem>* an attribute of type *EntryListener<LocalCIBuildJObQueueItem>* is shown, abstracting over the information that the objects stored will be of type of the inner class

pretense, where an operation *eat(food: Food)* is shown in the model, which does not occur in the source code.

4.2.2 Substitution

Substitutions are cases where a structure (most often a type) shown in the model is substituted/substitutes a different structure (type) in the source code. We found such substitutions for attributes, attribute types, parameters, parameter types, and return types. For example, Fig. 10a, b shows a case of **attribute substitution**, where an attribute *amount: BigDecimal* is shown in the model for the class *CheckingAccount*, instead of the attribute *expenseRepo: ExpenseRepository*, which is shown in the source code. For cases of a pure **attribute type substitution**, we find situations where non-primitive data types are substituted by other non-primitive data types and situations where non-primitive data types are shown in the model for attributes implemented by primitive data types. An example for a **parameter substitution** is the parameter *builder: DSSTPropagatorBuilder* of the method *addPropagationConfiguration(builder: DSSTPropagatorBuilder, provider: CovarianceMatrixProvider): SemiAnalyticalKalmanEstimatorBuilder* from class *SemiAnalyticalKalmanEstimatorBuilder* in project 8. It was shown as *propagator: DSSTPropagator* of the method *addPropagationConfiguration(propagator: DSSTPropagator, initialCovariance: CovarianceMatrixProvider): SemiAnalyticalKalmanEstimatorBuilder* in the model.

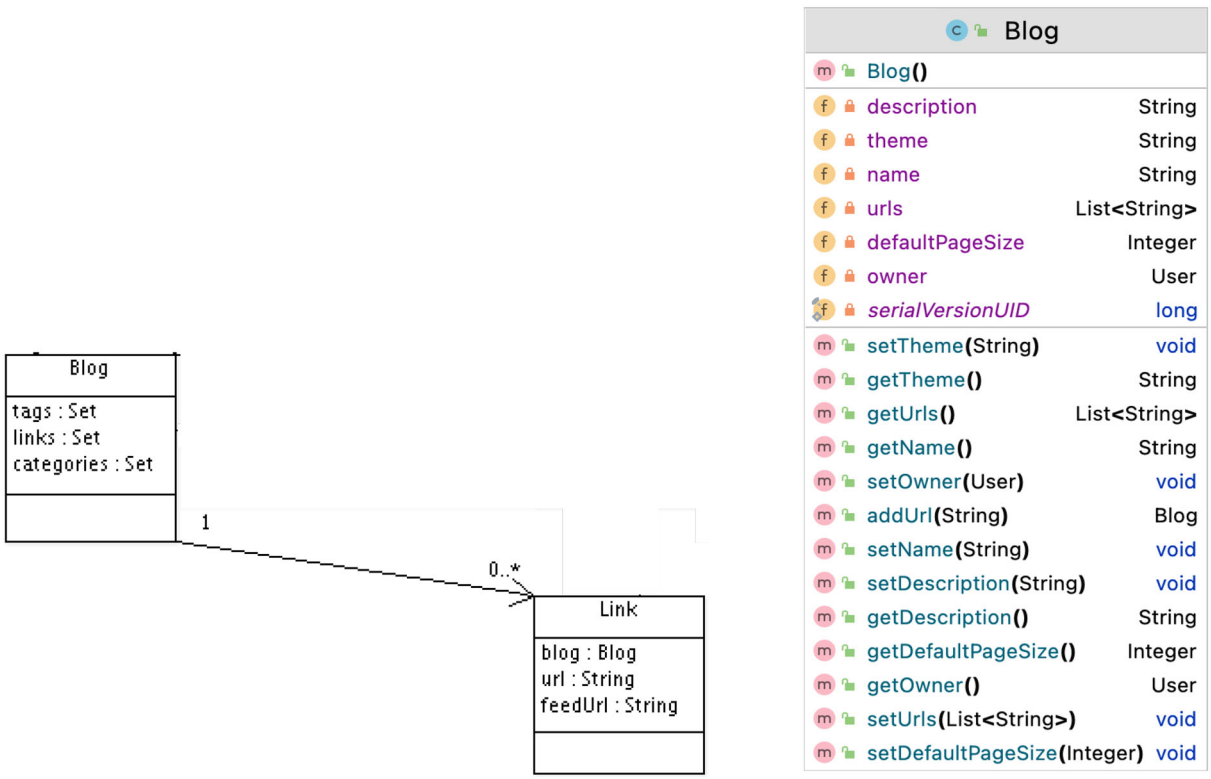
An example for a **return type substitution** is the method *modifyPet(): void* from class *Control* in project 2. Here the model shows a return type *Boolean* (*modifyPet(kit: Pet): Boolean*) instead of *void*.

4.2.3 Refactorings

Finally we observed two cases where the implementation and model differ in the way of simple refactorings, which concerned attributes and operations.

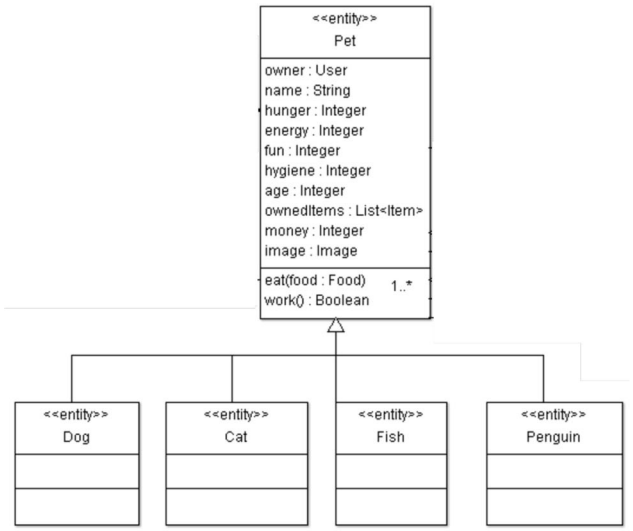
Table 5 Cases of disagreements

	Disagreements
Classes	
	Subclass pretense: The subclasses in the inheritance structures in the model are not present in the source code.
	Class pretense: Classes shown in the model are not present in the source code
Attributes	
	Attribute pretense: Attributes shown in the model are not present in the source code
	Attribute substitution: Attributes in the model are replaced by different attributes (with different names and types) in the source code
	Attribute type substitution: The type of an attribute in the source code is different from the type shown in the model
	Attribute pull up: Within an inheritance structure, attributes belonging to a superclass in the source code are shown as part of the subclasses in the model.
Operations	
	Operation pretense: Operations shown in the model are not present in the source code.
	Parameter pretense: Parameters shown in the model are not present in the source code
	Parameter substitution: Parameters in the source code and model are different.
	Parameter type substitution: A parameter has different types in the model and the source code
	Return type substitution: A method's return type in the source code and model are different
	Operation move: Operations are located in different classes in the source code and model
Relationships (between classifiers)	
	Relationship pretense: Relationships shown in the model are not present in the source code

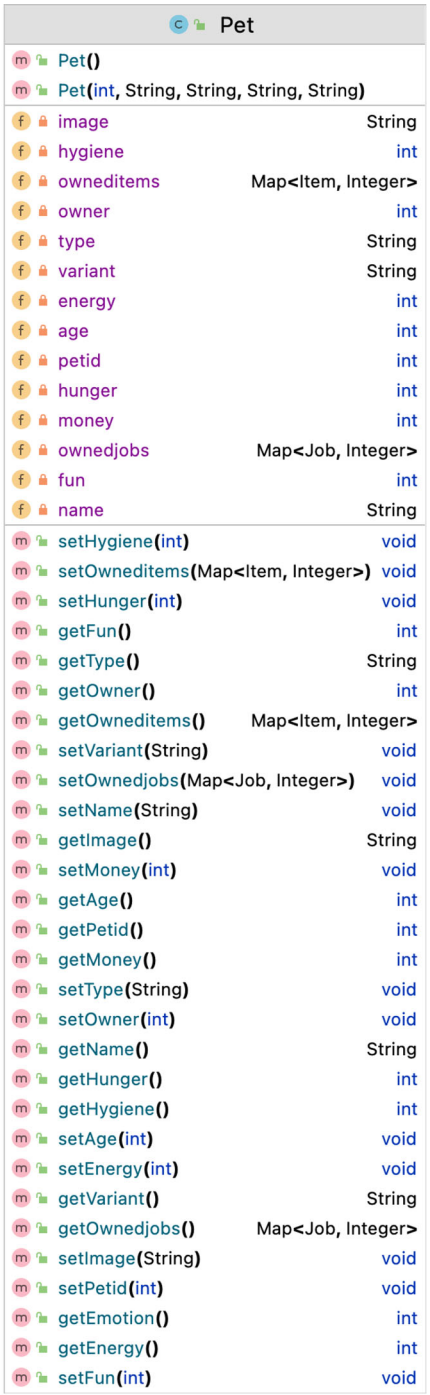


(a) Extract from the model of project 7 (re-layouted for readability). (b) Visualization of corresponding code part from project 7.

Fig. 8 Class pretense: The model shows a class *Link*. In the source code, the *Link* class is shown as an attribute *urls: List<String>* in the *Blog* class



(a) Extract from the model of project 2



(b) Visualization of corresponding code parts from project 2

Fig. 9 Subclass pretense: The subclasses of Dog, Cat, Fish, and Penguin of the inheritance structure with the superclass *Pet* and an operation *eat(food: Food)* in the model, which are both not present in the source code

(a) Extract from the model of project 3

(b) Visualization of corresponding code parts from project 3

Fig. 10 Attribute substitution: The model shows an attribute *amount: BigDecimal*, which is substituted by another attribute *expenseRepo: ExpenseRepository* in the source code

The first case is **attribute pull up**, where attributes are moved between superclass and subclasses within an inheritance structure. Figure 11a, b shows an example from project 2, where the attribute *image* is shown for the subclasses *Food* and *Upgrade* in the model. Within the source code the attribute is pulled up to the superclass *Item*. Note that this example also includes a case of **attribute type substitution** as the types of the attribute in the model (non-primitive type *image*) and the source code (primitive type *String*) are different.

The other case observed is **operation move**, where an operation is moved from one class to the other. For example, in the model of project 2, the operation *listUser(): Map* is shown in the class *DAO*. However, in the source code the operations is moved to the class *RaiseMeUp*. Similar to the other case of refactoring, also this refactoring is accompanied with other changes, namely an *attribute name inaccuracy* (see Sect. 4.3) concerning the methods name (*listUser()* vs. *listUsers()*) and a *parameter type omission* (as explained above).

Observation: Disagreements are characterized by the pretense of elements in the model that are not present in the code, substitution of elements, and refactorings between model and code.

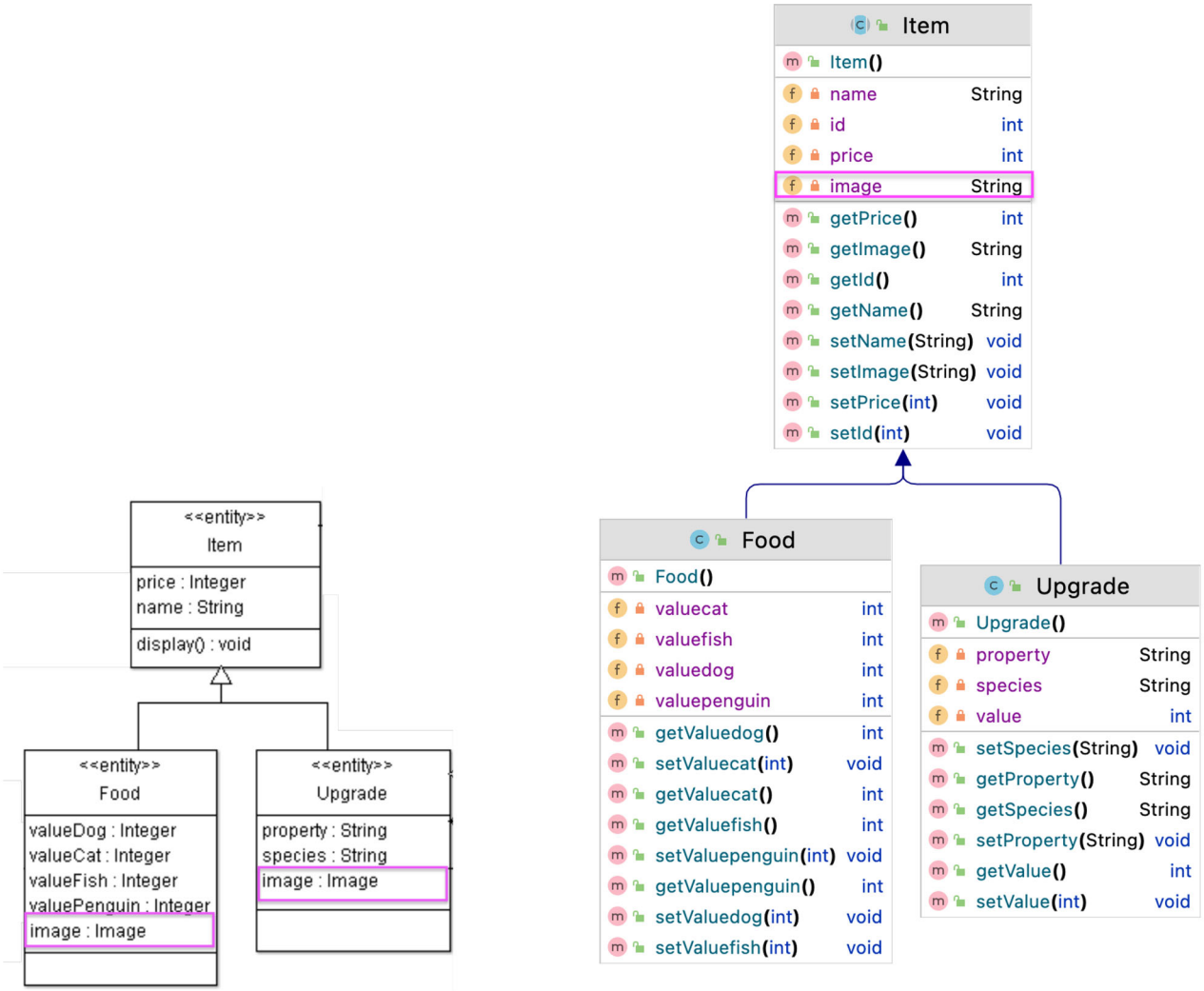
4.3 Inaccuracies

Table 6 summarizes identified inaccuracies, which come in two groups: name inaccuracies and type inaccuracies.

4.3.1 Name inaccuracies

Observed name inaccuracies concern class names, attribute names, operation names and parameter names. We identified the following situations.

Misspelling Sometimes the model includes a difference in spelling compared to the code. For example, in project 1 a class is named *SinglePlayModel* in the model, but *SinglePlayerModel* in the source code. Similarly in project 2, the



(a) Extract from the model of project 2.

(b) Visualization of corresponding code part from project 2.

Fig. 11 Attribute pull up: The attribute *image* that is shown for both subclasses *Food* and *Upgrade* in the model is pulled up to the superclass *Item* in the source code

method *onCreat* in class *SinglePlayer* has a parameter that is called *saveInstaceState* in the model and *saveInstanceState* in the source code. We observed such misspellings for class, operation, and parameter names.

Synonyms In some cases, names in the model and code are synonyms of each other. For example, the class *RegisterIncomeUI* in the source code of project 3 is called *IncomeRegisterUI* in the corresponding model. In project 4, the operation *getNoOfHours* from the source code is named *getNumberOfHours* in the model. We observed such synonyms for class, attribute, and operations names.

Renaming Some names are outright changed. So is the class *Control* from the model of project 2 called *RaiseMeUp* in the source code. Here the name in the model refers to the class's role within the Model-View-Controller pattern, while the name in the source code reflects the project's name. We observed such renaming for class and parameter names.

Conversion of case types In some cases, there is a case conversion, e.g., from *setbackground* in the model to *setBackground* in the code (operation name in project 1). We observed case types conversions for operation and attribute names.

Conversion from singular to plural We observed a case where the name of an operation changed from singular (*listFood*) in the model to plural (*listFoods*) in the source code (project 2). We observed this conversion from singular to plural only for operation names.

Table 6 Cases of inaccuracies

	Inaccuracies
Classes	
	Class name inaccuracy: A class has different names in the source code and model
Attributes	
	Attribute name inaccuracy: An attribute has different names in the source code and model
	Attribute type inaccuracy: An attribute type from the source code is inaccurately, but recognizably, shown in the model
Operations	
	Operation name inaccuracy: An operation has different names in the source code and model
	Parameter name inaccuracy: A parameter has different names in the source code and model
	Return type inaccuracy: A return type from the source code is inaccurately, but recognizably, shown in the model.
	Parameter type inaccuracy: A parameter type from the source code is inaccurately, but recognizably, shown in the model

4.3.2 Type inaccuracies

Types are also sometimes affected by inaccuracies. We found inaccuracies concerning attribute types and parameter types. For example, there is a nuance in Java about the difference between the primitive type *int* and the type *Integer* which represents a wrapper class for the primitive type *int*, allowing its instances to be treated like an object. We found a case where the attribute type *int* from the source code was represented as *Integer* in the model (attribute *money* in class *Pet* in project 2). In other cases, type inaccuracies just concern capitalization of type names, for example, the parameter type *char* was represented as *Char* in the model (class *SinglePlayModel* in project 1). This can be misleading since primitive types in Java are not capitalized, while the corresponding wrapper classes are. A case only concerning the capitalization of the return type name was found, for example, specifically for the segment of **CI**, the return type CompletableFuture<LocalCIBuildResult> from the code was represented as CompletableFuture<LocalCiBuildResult> in the model (class BuildJobManagementService in project 9).

> **Observation:** Inaccuracies in names are due to misspellings, synonym usage, renamings, as well as conversions of case types and singular/plural.

4.4 Summary of occurrences

Table 7 summarizes the found cases with regard to the projects that they have been found in. Not all cases have been observed in more than one project, while others occurred in most of the studied projects, namely attribute omission, operation omission, parameter name omission, return type omission, relationship omission, type parameter omission, class omission, parameter omission, subsystem omission, relationship pretense, class name inaccuracy and attribute name inaccuracy.

Similarly, we observe that most cases of disagreement are mostly due to one of the studied projects (project 2). Even though we found disagreements in 7 out of 9 projects, future work will need to show whether project 2 is an exception with regard to the variety of disagreements that have been found.

The outcome from the four projects in iteration 2 can confirm the taxonomy and cases obtained on the five projects from iteration 1. New occurrences were uncovered for most cases, namely operation omission, class omission, attribute omission, relationship omission, return type omission, parameter name omission, parameter omission, subsystem omission, default value omission, type parameter omission, relationship loosening, relationship pretense, attribute pretense, operation pretense, parameter pretense, attribute name inaccuracies, parameter name inaccuracies. Specifically, we found a new variety of instances for the case of type parameter omission, which can occur in parameter type (from iteration 2), attribute type (from iteration 2) and return type (from iteration 1). Five new difference types were found in iteration 2: class pretense, attribute type generalization, superclass omission, parameter substitution, and return type inaccuracy. For project 6, no occurrences were uncovered for disagreements and inaccuracies; in this case, we deemed that the developers were especially thorough in ensuring consistency between model and code.

> **Observation:** Iteration 2 largely confirms the findings from iteration 1, in particular, the taxonomy and list of cases, while adding new occurrences as well as five cases.

Table 7 Summary of occurrences of types of difference in the studied projects

Difference types	Affected projects	Occurrences
Real Abstraction		
Operation omission	1, 2, 3, 4, 5, 6, 7, 8, 9	51,949
Class omission	1, 2, 3, 4, 5, 6, 7, 8, 9	4905
Attribute omission	1, 2, 3, 4, 5, 6, 7, 8, 9	24,891
Relationship omission	1, 2, 3, 4, 5, 6, 7, 8	42
Return type omission	1, 2, 3, 4, 6, 8	67
Subsystem omission	3, 4, 5, 6, 7, 8, 9	7
Parameter name omission	1, 2, 3, 6, 9	74
Parameter omission	1, 2, 3, 4, 8	32
Relationship loosening	4, 6, 7	9
Default value omission	1, 2, 6	8
Type parameter omission	2, 3, 7, 8, 9	13
Subclass omission	5	25
Attribute type omission	4	15
Operation summary	2	6
Superclass omission	8	3
Attribute type generalization	9	3
Class summary	2	1
Attribute summary	2	1
Relationship summary	2	1
Disagreements		
Relationship pretense	2, 5, 7, 8, 9	20
Operation pretense	2, 4, 8	31
Attribute pretense	2, 7, 9	6
Attribute type substitution	2	6
Operation move	2	5
Subclass pretense	2	4
Attribute substitution	3	2
Parameter pretense	2, 9	2
Class pretense	7	1
Parameter substitution	8	1
Parameter type substitution	2	1
Return type substitution	2	1
Attribute pull up	2	1
Inaccuracies		
Class name inaccuracy	1, 2, 3, 5	5
Operation name inaccuracy	1, 2, 4	19
Attribute name inaccuracy	1, 2, 4, 8, 9	17
Attribute type inaccuracy	2, 3	16
Parameter name inaccuracy	1, 2, 8	9
Parameter type inaccuracy	1	1
Return type inaccuracy	9	1

5 Discussion

We now discuss the implications of our study to researchers, practitioners, and teachers. First, we discuss our contribution to theory building in the research line on human abstraction, including how our results relate to previous findings. Second, we take a closer look at selected detailed observations and their potential implications. Third, we discuss the potential use of our work in teaching. Fourth, we present our thoughts on potential follow-up research on best practices for modeling as we as the development of new AI-based reverse engineering approaches.

5.1 Theory building

Our findings partially support findings and design decisions from previous studies on abstraction, such as those introduced in Sect. 2.1. For example, our results confirm that classes may be omitted, e.g., if they are view-related or are utility classes [17, 27]. This is in line with the broader principle of filtering irrelevant information [3]. However, our results also partially lead to new findings and add new nuances to existing ones. Specifically, we identified the following four insights about abstraction.

Too detailed modeling? We indeed found disagreement as a source of differences in seven out of nine considered projects, which supports previous developer studies that found supposedly deliberate deviations made by developers [6, 7]. We were surprised to observe this disagreement to be so widespread in our relatively small sample. It is also remarkable that four of the nine projects are hit especially hard by this, with multiple occurrences and types of disagreements each. One hypothesis is that the modeling was too detailed and did not leave enough space for developers to make decisions. An immediate *lesson learned* that practitioners can take from those observations is to reflect on what aspects need to be specified already during modeling. *Future work* will need to investigate what projects are most at risk of performing too detailed modeling and what the impact of disagreements on the usefulness of models (e.g., as documentation) are.

No systematic abstraction by type of modeling element! Even though we found omissions of attributes, only in one case they were systematically omitted. Thus, in contrast to Baltes and Diehl [27], we do not find a systematic trend to omit attributes. This might be explained, by the fact that they studied sketches, which are different in nature from models committed to repositories. Thus, for models committed to repositories, omission of attributes much more likely follows semantic considerations. It is currently unknown whether the choice of what attributes to show depends on the relevance of the attribute for comprehension of the domain, on the modeler's perception of what attributes are mandatory to be provided by the implementation, or on other reasons. A *lessons learned* here is that we should teach students to make a deliberate decision whether to include all, none, or a subset of the attributes, depending on the intended usage of the diagram. In *future work* we need to better understand reasons for showing and omitting attributes and other model elements, as according to our findings, omission of model elements is the most important tool for abstraction.

Abstraction by summary is rare. Particularly surprising was that summary elements, as they are proposed by approaches to create abstraction automatically [12, 30] and which seems in line with the *mapping out problem structures* process described by Qian and Choi [3], were rarely observed in our subject projects, namely only in one out of nine cases, which is not one of the two large systems studied. Thus, we hypothesize that in practice abstraction by summary is rarely done in class diagrams. *Future work* will need to confirm or refute this hypothesis.

Focus over-abstraction? When creating a model of a system, especially for larger systems, there are two basic options. Option one is to create an abstraction over the whole system. This can be done by only showing core classes of the overall system or by showing a high-level structure that summarizes multiple code parts into modeled classes (see discussion point above) or components. Option two is to instead focus on one or more subsystems of the system only, omitting the rest. In this study, we observed for seven out of the nine systems that option 2 is chosen and abstraction is reached by omitting subsystems. The two systems where this is not the case are on the smaller side of systems studied here, in terms of classes. Obviously, our choice to study class diagrams might cause a bias in our findings toward cases that use option two. Regarding option 1, we have during the creation of the Lindholmen dataset [21] indeed seen very high-level component diagrams. However, in many of these cases, the component diagrams showed simple architectural patterns, such as the model-view-controller pattern, only. Thus, there is a need to study the question of whether abstraction to high-level architectures is common and, if so, what it looks like. We hope to address this in *future work*.

Abstraction principles scale? We found that the difference types identified in the first iteration were surprisingly suitable to classify the differences found in the two bigger projects 8 and 9. In fact, we only identified 4 new difference types when studying these bigger systems (2 types of abstraction, 1 disagreement, and 1 type of inaccuracy). This leads to the hypothesis that the identified abstraction types might indeed scale for class diagrams in systems of all sizes. Future work is needed as 2 is still a fairly small number and not enough to represent all medium and large-sized systems.

5.2 Detailed observations and implications

Real abstraction vs. disagreement. Due to their definitions, there is a symmetry between real abstraction and disagreement. Indeed some types of differences, such as, e.g., the omission of elements of an inheritance structure (subclasses) vs. the pretense of elements of an inheritance structure (subclasses), can be considered dual opposites of each other. Such omission-pretense pairs exist also for attributes, operations, parameters, and relationships. However, this does not work for all cases of omission, as source code leaves much less room for underspecification than models do. For example, even though we found a case of return type omission, the dual phenomenon *return type pretense* would not be possible. In the first phase of this work, which was presented in the earlier conference paper [26], this allowed us to speculate and predict that the dual opposites of some of the found cases might exist in practice, even though we did not observe them in the studied systems, specifically, *class pretense* arising as the dual opposite of *class omission*. In our second phase, after considering new projects, we were able to confirm that class pretense indeed exists in practice.

Name inaccuracies. In the *inaccuracies* category, a particularly widespread issue was with naming inaccuracies, which we found in class, attribute, operation, and parameter names, with several different explanations. Naming inaccuracies are crucial to consider for developers of tools that need to automatically match models to code, in particular, reverse engineering tools that update existing models based on code, and consistency checking tools. These tools often work based on names [18, 19]. They also raise a concern about training abstraction capabilities in tools on real examples: as inaccuracies are a potential source of confusion, tools should strive to avoid producing them, which requires careful curation of training data. Finding additional types of reasons for naming accuracies is also a relevant area for future research: as we found so many instances of this issue in only a few cases, we speculate that we have not reached saturation, and could even find more different issues in further projects.

5.3 Potential use of results and dataset in teaching.

Abstraction is inherently hard to teach [54]. Two issues that students struggle with are the motivation for abstraction, as students may deem abstractions done in modeling as too far away from programming [55], and understanding the mechanics of abstraction, which might be unfamiliar to them due to their earlier experiences in low-level programming tasks [56]. Our results and dataset could contribute to addressing both issues: Toward improving motivation, we highlight nine open-source systems in which developers actually relied on modeling and abstraction to perform development tasks, together with sizable code bases that were developed with these models. These could be introduced as educational examples in class, where they would allow students to understand abstraction phenomena in a rich real-life environment, facilitating realism in teaching [56]. Toward aiding in understanding abstraction mechanics, our detailed results shed light on questions of how abstraction works in practice, in the context of class diagrams—which kind of information is usually, typically, and rarely omitted, and which information might be altered in abstractions. Our illustrative cases as presented in the paper could be included in teaching materials on this topic.

5.4 Follow-up research

On abstraction ability of AI-based tools. AI systems, more specifically, large language models (LLMs), have recently began to unfold a significant potential for model-driven engineering tasks [57]. At the same time, there is a reason to stay cautious, as LLMs have been heavily criticized for their general lack of reliability and focus on producing plausible-sounding, rather than accurate, solutions [58]. In the context of this research, on abstraction in modeling, the most promising application area of AI is to support automated reverse engineering tools, where the ability to create human-like abstractions would be a significant step forward over the state-of-the-art.

In the context of developing AI-based reverse engineering tools, we foresee the following uses of our work: First, for design activities: specifically, the performance of LLMs has been observed to significantly depend on the quality of prompts, making *prompt engineering* the most significant activity for applying an LLM for a given task. By including information from our abstraction taxonomy in the used prompt, LLMs can be guided to produce solutions that incorporate common abstraction types. Second, for evaluation activities: evaluating the ability of an LLM-based approach to produce human-like solutions requires high-quality, annotated datasets describing exactly how humans abstract in specific systems. Our dataset with a total of 85 295 code elements could be a useful resource for this task. An open question to be addressed in this context is how to deal with the fact that there might not be *one single valid* abstraction for a given system, but multiple ones. A possible strategy is to guide the considered tools with hints to produce a solution that looks a close as possible to the extended one, and then evaluate the ability to do so, using a similarity measurement.

On best practices for modeling. Our findings are descriptive in nature and hence, should not be directly considered as best practices for modeling. Still, they could inform follow-up research dedicated to the identification of best practices. Relevant questions are *if* and *under which conditions* applying our abstraction cases leads to an improvement in relevant quality aspects, such as the readability of a class diagram

or the maintainability of the system when using the class diagram as a reference. These questions are empirical in nature and can be answered in the form of relevant empirical methods, such as interviews and controlled experiments (e.g., letting different developers solve the same task by using different reference class diagrams).

On project contributor dynamics. In this paper, we did not investigate the different roles of different and how they interact with abstractions, for example, by studying the question of who *creates*, *updates*, and *uses* abstractions in what context. Follow-up work in this direction would be worthwhile and could directly benefit from our provided dataset, as we recorded the change information together with commit IDs that allow to retrieve names of involved committers.

6 Threats to validity

Internal Validity To avoid possible misconceptions arising from subjective definitions of the term "differences", we used the specifications for UML v2.4.1 [59] and JavaSE8 [60], which were published closest to the time of the start of the projects, to obtain clear definitions for considered model and code elements and derive precise mappings between them.

Another threat is that during data selection if multiple code versions correspond to the considered model, we make a (motivated) selection between them. Furthermore, the assumption that the studied class diagram shows the complete model might not always be true. In that case, we would overestimate the abstraction regarding the complete model. Nonetheless, the results are valid when it comes to abstraction decisions about what to show in the diagram. This leaves it possible that, e.g., missed attributes and operations in a particular code version are implemented in other code versions. Yet, our finding that a difference existed at a given point in time remains valid. Furthermore, there is a risk that some of the captured differences are due to code evolution. With our data selection method, we aimed to minimize this risk.

In addition, there is a threat that we might have selected projects where the code has been automatically generated based on the models. During project selection, we took care to avoid projects where we could identify that this was the case. Of course, cannot completely rule out that we did not identify an instance of generated code. However, our results indicate that we were to some extent successful, due to the many inaccuracies found, which one would not expect in cases with generated code (if generated from the studied diagrams).

Finally, we cannot exclude that some of the cases we deem as deliberate abstractions and disagreements are, in fact, oversights. For example, an alternative explanation for instances of *attribute omission* is that the designer forgot to include the attribute in the class diagram. The information presented in Table 4 mitigates this issue to some extent—difference types that occurred in multiple projects and tens or hundreds of cases seem likely to be intentional in at least a part of them.

External validity. This paper focuses only on Java projects. Therefore, future studies are required to conclude whether our results are valid for systems built with other languages as well. Our results are most likely to generalize to languages that are similar to Java. This applies especially to object-oriented languages that have Java-like concepts of classes, methods, attributes, and relations, such as C# and C++. More broadly, they might generalize to languages that have a notion of *datatypes*, to the extent this notion can be aligned with these concepts.

Furthermore, as with all research that is conducted with GitHub repositories, there is a risk that the results are not representative of the industrial use of models. Specifically, five out of the nine projects have a fairly short active period and all except two projects have below 10 contributors. Given the diversity of considered projects, we do not see an obvious issue that prohibits the generalization of our results to industrial contexts. Still, understanding to which extent studies conducted on open-source projects can support conclusions that generalize to industrial projects is an intricate research question that requires dedicated studies.

We also did not assess the quality of the selected projects, which means that we cannot make statements on representativeness for low- or high-quality projects. We tried to mitigate this risk by scanning the selected projects for obvious signs that they might stem from, e.g., teaching materials or classroom projects. Still, future work on industrial projects is required to establish generalizability of our results.

Our study does not distinguish between different abstraction levels that can be targeted by a class diagram (e.g., how close should the diagram be to the implementation?). This might affect the generalizability of specific findings to other contexts, in the form that some of the identified abstraction cases might not be applicable in these contexts. Finally, we focused on class diagrams stored in image formats. It is possible that models stored as .uml or .xmi relate differently to the source code. Also here future studies are required to further explore how abstraction changes, e.g., if modeling tools change.

7 Conclusion

In this paper, we conducted a thorough examination of manual abstraction by analyzing nine open-source software projects, focusing on their class models and codebases. We systematically created a set of mappings between model and code elements, which we analyzed to identify three primary types of discrepancies-real abstraction, disagreements, and inaccuracies-along with corresponding cases. These discrepancies highlight significant practices in manual abstraction,

such as *subclass and superclass omissions*, *type parameter omission*, and *relationship loosening*.

We suggest several directions for future research. First, our results could inform the development and assessment of advanced reverse engineering tools, consistency-checking rules, and manual abstraction guidelines. It would be particularly interesting to explore how the types of manual abstraction we identified align with those created using large language models. Second, future research should explore manual abstraction in various diagram types beyond class models. Our overarching categories of omission, pretense, and inaccuracies are not exclusive to class diagrams, and it would be valuable to investigate their applicability to other diagram types. Third, our discussion of theory building aspects leads to several new research questions to be investigated in future work, such as: *What makes projects prone to performing too detailed modeling? What is the impact of disagreements on the usefulness of models (e.g., as documentation)? Why do developers omit some elements, such as certain attributes, rather than others? How common is high-level abstraction of architecture models, beyond architectural styles, and what does it look like?* With our taxonomy and comprehensive list of cases, our contribution paves the way for future studies addressing these questions.

Funding Open access funding provided by University of Gothenburg.

References

1. Dennis, A., Wixom, B.H., Roth, R.M.: Systems Analysis and Design (2008)
2. Wing, J.M.: Computational thinking and thinking about computing. Philos. Trans. R. Soc. A: Math. Phys. Eng. Sci. **366**(1881), 3717–3725 (2008)
3. Qian, Y., Choi, I.: Tracing the essence: ways to develop abstraction in computational thinking. Educ. Tech. Res. Dev. **71**(3), 1055–1078 (2023)
4. Vogel-Heuser, B., Fay, A., Schaefer, I., Tichy, M.: Evolution of software in automated production systems: challenges and research directions. J. Syst. Softw. **110**, 54–84 (2015)
5. Osman, M.H., Chaudron, M.R.V.: UML usage in open source software development: A field study. In: 3rd International Workshop on Experiences and Empirical Studies in Software Modeling Co-located with 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS), pp. 23–32 (2013)
6. Gueheneuc, Y.-G.: A systematic study of UML class diagram constituents for their abstract and precise recovery. In: 11th Asia-Pacific Software Engineering Conference, pp. 265–274 (2004)
7. Ho-Quang, T., Hebig, R., Robles, G., Chaudron, M.R.V., Fernandez, M.A.: Practices and perceptions of UML use in open source projects. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), pp. 203–212 (2017)
8. Bruneliere, H., Cabot, J., Dupé, G., Madiot, F.: Modisco: a model driven reverse engineering framework. Inf. Softw. Technol. **56**(8), 1012–1032 (2014)
9. Koschke, R.: Architecture reconstruction: Tutorial on reverse engineering to the architectural level. Software Engineering: International Summer Schools, ISSSE 2006-2008, Salerno, Italy, Revised Tutorial Lectures, pp. 140–173 (2009)
10. Thung, F., Lo, D., Osman, M.H., Chaudron, M.R.V.: Condensing class diagrams by analyzing design and network metrics using optimistic classification. In: Proceedings of the 22nd International Conference on Program Comprehension. ICPC 2014, pp. 110–121 (2014)
11. Egyed, A.: Automated abstraction of class diagrams. ACM Trans. Softw. Eng. Methodol. (TOSEM) **11**(4), 449–491 (2002)
12. Egyed, A.: Semantic abstraction rules for class diagrams. In: Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering, pp. 301–304 (2000). IEEE
13. Booshehri, M., Luksch, P.: Condensation of reverse engineered UML diagrams by using the semantic web technologies. In: Proceedings of the International Conference on Information and Knowledge Engineering (IKE), p. 95 (2015)
14. Peldszus, S., Strüber, D., Jürjens, J.: Model-based security analysis of feature-oriented software product lines. In: ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, pp. 93–106 (2018)
15. Priefer, D., Rost, W., Strüber, D., Taentzer, G., Kneisel, P.: Applying mdd in the content management system domain. Softw. Syst. Model. **20**, 1919–1943 (2021)
16. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al.: Language models are few-shot learners. Adv. Neural. Inf. Process. Syst. **33**, 1877–1901 (2020)
17. Osman, M.H., Zadelhoff, A., Stikkolorum, D.R., Chaudron, M.R.V.: Uml class diagram simplification: What is in the developer's mind? EESSMod '12 (2012)
18. Antoniol, G., Caprile, B., Potrich, A., Tonella, P.: Design-code traceability recovery: selecting the basic linkage properties. Sci. Comput. Program. **40**(2), 213–234 (2001). (**Special Issue on Program Comprehension**)
19. Opzeeland, D.J., Lange, C.F., Chaudron, M.R.V.: Quantitative techniques for the assessment of correspondence between UML designs and implementations. In: 9th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (2005)
20. Shatnawi, R., Alzu'bi, A.: A verification of the correspondence between design and implementation quality attributes using a hierarchal quality model. IAENG Int. J. Comput. Sci. **38**(3), 225–233 (2011)
21. Hebig, R., Quang, T.H., Chaudron, M.R.V., Robles, G., Fernandez, M.A.: The quest for open source projects that use UML: Mining github. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems. MODELS '16, pp. 173–183 (2016)
22. Dabic, O., Aghajani, E., Bavota, G.: Sampling projects in GitHub for MSR studies. In: 18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, pp. 560–564 (2021). IEEE
23. Berardi, D., Calvanese, D., De Giacomo, G.: Reasoning on UML class diagrams. Artif. Intell. **168**(1), 70–118 (2005)

24. Guéhéneuc, Y.-G.: A reverse engineering tool for precise class diagrams. In: Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research, pp. 28–41 (2004)
25. The authors: Replication Package for 'An Empirical Study of Manual Abstraction between Class Diagrams and Code of Open Source Systems'. https://osf.io/p4jdr/?view_only=34b153d3b7704f4a84befd661dc9c6a1
26. Zhang, W., Zhang, W., Strüber, D., Hebig, R.: Manual abstraction in the wild: A multiple-case study on oss systems'class diagrams and implementations. In: MODELS'23: International Conference on Model-Driven Engineering Systems and Languages (2023)
27. Baltes, S., Diehl, S.: Sketches and diagrams in practice. FSE **2014**, 530–541 (2014)
28. Müller, H.A., Jahnke, J.H., Smith, D.B., Storey, M.-A., Tilley, S.R., Wong, K.: Reverse engineering: a roadmap. In: Proceedings of the Conference on the Future of Software Engineering, pp. 47–60 (2000)
29. Chikofsky, E.J., Cross, J.H.: Reverse engineering and design recovery: a taxonomy. IEEE Softw. **7**(1), 13–17 (1990)
30. Guizzardi, G., Figueiredo, G., Hedblom, M.M., Poels, G.: Ontology-based model abstraction. In: 2019 13th International Conference on Research Challenges in Information Science (RCIS), pp. 1–13 (2019). IEEE
31. Osman, M.H., Chaudron, M.R.V., Van Der Putten, P.: An analysis of machine learning algorithms for condensing reverse engineered class diagrams. In: 2013 IEEE International Conference on Software Maintenance, pp. 140–149 (2013). IEEE
32. Yang, X., Lo, D., Xia, X., Sun, J.: Condensing class diagrams with minimal manual labeling cost. In: 2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC), vol. 1, pp. 22–31 (2016). IEEE
33. Riedl-Ehrenleitner, M., Demuth, A., Egyed, A.: Towards model-and-code consistency checking. In: 2014 IEEE 38th Annual Computer Software and Applications Conference, pp. 85–90 (2014). IEEE
34. Chavez, H.M., Shen, W., France, R.B., Mechling, B.A., Li, G.: An approach to checking consistency between UML class model and its Java implementation. IEEE Trans. Softw. Eng. **42**(4), 322–344 (2015)
35. Jongeling, R., Fredriksson, J., Ciccozzi, F., Cicchetti, A., Carlson, J.: Towards consistency checking between a system model and its implementation. In: Systems Modelling and Management: First International Conference, ICSMM 2020, Bergen, Norway, June 25–26, 2020, Proceedings 1, pp. 30–39 (2020). Springer
36. Jongeling, R., Cicchetti, A., Ciccozzi, F., Carlson, J.: Towards boosting the openmbee platform with model-code consistency. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, pp. 1–5 (2020)
37. Jongeling, R., Fredriksson, J., Carlson, J., Ciccozzi, F., Cicchetti, A.: Structural consistency between a system model and its implementation: a design science study in industry. J. Object Technol. **21**(3), 3–1 (2022)
38. Baltes, S., Diehl, S.: Sketches and diagrams in practice. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 530–541 (2014)
39. Karasneh, B., Chaudron, M.R.: Online img2uml repository: An online repository for UML models. In: EESSMod@ MoDELS, pp. 61–66 (2013)
40. Stol, K.-J., Fitzgerald, B.: The abc of software engineering research **27**(3) (2018)
41. Robles, G., Ho-Quang, T., Hebig, R., Chaudron, M.R.V., Fernandez, M.A.: An extensive dataset of UML models in GitHub. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pp. 519–522 (2017). IEEE
42. Romeo, J., Raglianti, M., Csaba, N., Lanza, M.: UML is back. or is it? In: ICSE 2025 47th International Conference on Software Engineering (2025)
43. Osman, M.H., Ho-Quang, T., Chaudron, M.R.V.: An automated approach for classifying reverse-engineered and forward-engineered UML class diagrams. In: 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 396–399 (2018)
44. GitHub - ZooTypers. https://github.com/orgs/ZooTypers/repositories Accessed 12 Dec 2022
45. GitHub - lekogabi/RaiseMeUp. fork from original. https://github.com/WenliZhang1102/RaiseMeUp Accessed 12 Dec 2022
46. GitHub - AntonioRochaOliveira/EAPLI_PL_2NB. https://github.com/AntonioRochaOliveira/EAPLI_PL_2NB Accessed 12 Dec 2022
47. GitHub - fmacicasan/FreeDaysIntern. https://github.com/fmacicasan/FreeDaysIntern Accessed 12 Dec 2022
48. GitHub - tekosds/NeurophChanges. https://github.com/tekosds/NeurophChanges Accessed 12 Dec 2022
49. GitHub - PatrickFromThe29/PatrickFromTheMOO. https://github.com/PatrickFromThe29/PatrickFromTheMOOC Accessed 12 Dec 2022
50. GitHub - jdkcn/myblog. https://github.com/jdkcn/myblog Accessed 12 Dec 2022
51. GitHub - CS-SI/Orekit. https://github.com/CS-SI/Orekit Accessed 12 Oct 2024
52. GitHub - ls1intum/Artemis. https://github.com/ls1intum/Artemis Accessed 12 Oct 2024
53. Mu, H., Jiang, S.: Design patterns in software development. In: 2011 IEEE 2nd International Conference on Software Engineering and Service Science, pp. 322–325 (2011)
54. Brosch, P., Kappel, G., Seidl, M., Wimmer, M.: Teaching model engineering in the large. In: MODELS Educators Symposium (2009)
55. Ciccozzi, F., Famelis, M., Kappel, G., Lambers, L., Mosser, S., Paige, R.F., et al: How do we teach modelling and model-driven engineering? a survey. In: MODELS Educators Symposium, pp. 122–129 (2018). ACM
56. Strüber, D.: The complexity paradox: An analysis of modeling education through the lens of complexity science. In: 2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 94–97 (2023). IEEE
57. Combemale, B., Gray, J., Rumpe, B.: Large language models as an"operating"system for software and systems modeling. Softw. Syst. Model. **22**(5), 1391–1392 (2023)
58. Hicks, M.T., Humphries, J., Slater, J.: Chatgpt is bullshit. Ethics Inf. Technol. **26**(2), 38 (2024)
59. Object Management Group, Inc.: About the unified modeling language specification version 2.4.1. Accessed 27 March 2024
60. Oracle Corporation: The Java^{TM} Tutorials. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html Accessed 05 Jan 2024

Wenli Zhang received her master's degree in Software Engineering from Chalmers University of Technology, Sweden, in 2023 and bachelor's degree in Software Engineering from South-Central University for Nationalities, China, in 2018. She is a software engineer, focusing on delivering automation solutions to clients. In her master's thesis, she conducted an empirical analysis to investigate the characteristics of manual abstraction between class diagrams and source code in open-source systems.

Weixing Zhang is a Ph.D. student at the Department of Computer Science and Engineering at the University of Gothenburg, Sweden, which is also affiliated with Chalmers University of Technology. His research areas are model-driven engineering and domain-specific languages. He received his Master of Engineering degree from Beijing Jiaotong University, China, in 2013, and then worked as a software engineer in China's industry for 7 years. He has extensive and industrial software development experience, and his technical stack involves various fields, e.g., C/C++/C#/Java/Python, VxWorks, Windows, Linux, etc. In his current PhD, he has published several academic papers in journals or conferences including SoSyM, MODELS, SEAA, SLE, etc.

Daniel Strüber is an associate professor at Chalmers and University at Gothenburg, Sweden, and an assistant professor at Radboud University in Nijmegen, the Netherlands. His research interests are in model-driven engineering, AI engineering, and variant-rich systems. He was awarded his doctoral degree from Philipps University Marburg, Germany, and worked as a post-doctoral researcher at University of Koblenz and Landau, Germany, and Gothenburg University, Sweden. He is a co-author of over 100 papers with six Best Paper Awards. He has been a Program Committee member of several leading conferences, including ICSE, ASE, MODELS, and a Program Chair of premier community venues, such as SPLC and ICGT.

Regina Hebig is a professor for Software Engineering at the University of Rostock, Germany. Her research interests are in model-driven engineering, software evolution and the use of AI for software engineering. She received her doctoral degree from the University of Potsdam, Germany. She worked as post-doctoral researcher at the Sorbonne University (Pierre-et-Marie-Curie), Paris, France, and at the University of Gothenburg, Sweden, as assistant and later associate professor between 2015 and 2023. She served as Program Committee member for several leading conferences, including MODELS, FSE, and ICMSE, and as Program Chair of venues such as SLE and ICSSP.