Manual Abstraction in the Wild: A Multiple-Case Study on OSS Systems' Class Diagrams and Implementations

Wenli Zhang [1], Weixing Zhang [2], Daniel Strüber [2,3], Regina Hebig [4]
[1]*Chalmers University of Technology, Gothenburg, SE* [2]*Chalmers | University of Gothenburg, Gothenburg, SE*
[3]*Radboud University, Nijmegen, NL* [4]*University of Rostock, Rostock, DE*
wenliz@student.chalmers.se, {weixing,danstru}@chalmers.se, regina.hebig@uni-rostock.de

***Abstract*—Models are a useful tool for software design, analysis, and to support the onboarding of new maintainers. However, these benefits are often lost over time, as the system implementation evolves and the original models are not updated. Reverse engineering methods and tools could help to keep models and implementation code in sync; however, automatically reverse-engineered models are typically not abstract and contain extensive information that prevents understanding. Recent advances in AI-based content generation make it likely that we will soon see reverse engineering tools with support for human-grade abstraction. To inform the design and validation of such tools, we need a principled understanding of what manual abstraction is, a question that has received little attention in the literature so far.**

Towards this goal, in this paper, we present a multiple-case study of model-to-code differences, investigating five substantial open-source software projects retrieved via repository mining. To explore characteristics of model-to-code differences, we, all in all, manually matched 466 classes, 1352 attributes, and 2634 operations from source code to 338 model elements (classes, attributes, operations, and relationships). These mappings precisely capture the differences between a provided class diagram design and implementation codebase. Studying all differences in detail allowed us to derive a taxonomy of difference types and to provide a sorted list of cases corresponding to the identified types of differences. As we discuss, our contributions pave the way for improved reverse engineering methods and tools, new mapping rules for model-to-code consistency checks, and guidelines for avoiding over-abstraction and over-specification during design.

***Index Terms*—software design, modeling**

I. Introduction

Models are an important tool for software analysis and design [1]. By capturing structural and behavioral system aspects at a suitable abstraction level, models aid with thinking, communication, and as a blueprint for implementing software via manual programming and automated code generation.

However, once that the development of a system has moved to implementation and maintenance, models exist as a separate artifact from the implementation code, which leads to challenges with keeping models and code synchronized with each other [2]. The literature reports on two main sources of model-to-code inconsistencies: First, during maintenance, developers tend to neglect to update the model after code changes [3]. Second, already during implementation, developers may either miss or intentionally deviate from parts of the model [4], [5]. Once that models and code become inconsistent, models become less useful and, worse, can even create confusion when software engineers try to use them for system comprehension.

A solution could be offered by available reverse engineering methods and tools, which can automatically extract models from code. Still, as traditional reverse engineering tools usually lack an ability to abstract, their produced models tend to contain excessive information, rendering them ineffective for comprehension. Recent advances in AI-based content generation, specifically, large language models such as GPT-4 [6], suggest an ability to mimic human-grade abstraction. As such, they could enable the development of improved reverse engineering tools with native support for abstraction. Yet, to design and validate such techniques, we need an understanding of what abstraction actually entails. Understanding the characteristics of manual abstraction requires in-depth, manual studies.

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 [7], 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 [8]–[10]. 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 five substantial cases with available models and implementation code, retrieved from the Lindholmen Dataset [11]. 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 [12], 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 [13], 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 the main cause of omission on class level is *inheritance structure 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 *collection type underspecification* and *relationship loosening*, acknowledging that collection types 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 [14], which includes links to the models, code versions used, manually annotated models, reverse-engineered models, and corresponding comparison templates for the five cases.

II. Related Work

This section focuses on studies of abstraction, reverse engineering, and the consistency of models and code.

a) 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 [7]. 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 [7]. 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*, and *s*hould be excluded from class diagrams [7]. 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 [15]. What the above studies have in common is that they are based on participant opinions and experiences rather than studying actual models.

b) Reverse-Engineering Class Diagrams: Müller et al. [16] defined reverse engineering in [17] 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. [13] and infers relationships in class diagrams. A well-known tool is MoDisco [18], which is a framework for model-driven reverse engineering that extracts information from existing artifacts to generate different representations of the system. Koschke [19] 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 [20], approaches for abstraction by rules and using machine learning were developed. In the former group, Egyed [21] [22] defined semantic abstraction rules, such as a rule to substitute two relationships that form an indirect relationship with one direct relationship. Booshehri and Luksch [23] 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. [24]. 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 [25]. Thung et al. extended the work of Osman et al. by adding network metrics (e.g., closeness centrality) and achieved a 9% improvement [20]. Yang et al. [26] 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.

c) Consistency Check(s) between Code and Design: Reverse-engineered diagrams are generated by extracting information from code based on the absence of a design that can be referenced. In contrast, consistency checks between code and design rely on the existence of an existing design. Multiple semantics can explain the same piece of code, which makes evolving designs of higher quality provided traceability with code is maintained [8]. Different models and methods exist to check consistency between code and design, with Antoniol et al. [8] and Dennis et al. [9] finding that class names play the most critical role in mappings between entities in design and

constructs in code. However, a lack of systematic study exists on the reasons for deviations between unmatched classes.

III. Methodology

We performed a multiple-case study [27] 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?

A. Project Selection

The Lindholmen dataset [11] is 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 [11], [28]. 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.

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, it is required to study multiple projects as we do not expect to get a full picture of typical differences by studying a single project only. In the end, we decided to study five 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. 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.

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. Due to the low success rate of this approach, we did not perform it exhaustively. Nevertheless, this strategy lead 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, we narrowed down our preselection, by using an available list with class diagrams from the Lindholmen dataset for which an image was available [29]. This list contained 415 class diagrams identified as forward diagrams, whose associated projects we considered exhaustively. This lead to the identification of two further projects (FreeDaysIntern, NeurophChanges), which were indeed the only remaining ones after the filter criteria were considered. The search was terminated when the five projects have been identified. While none of our considered projects is actively developed at this point, our projects span a range of active project duration, between 1,5 and 28 months of activity. Table I[1] summarizes these five projects. 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.

B. Selection of Model and Source Code 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 project 3, 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.

1) Selection aim: Selecting a version 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.

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. We first assume that the version of source code to select is likely to be close to the commit with which the selected model was committed. So

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

TABLE I: The studied projects (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; model elements = the number of classes, attributes, operations, and relationships shown in the studied model

ID	Name	Domain	Active period	#versions	#contributors	files	source code			model elements
							classes	operations	attributes	
1	ZooTypers [30]	Android	∼1.5	745	5	87	15	77	52	59
2	RaiseMeUp [31]	GUI	∼1.5	24	2	168	40	545	474	141
3	EAPLI_PL_2NB [32]	Web applications	∼2	483	9	103	60	255	51	54
4	FreeDaysIntern [33]	Web applications	∼19	449	3	302	92	502	216	65
5	NeurophChanges [34]	GUI	∼28	534	7	2270	259	1255	559	19

we select that version of source code as a starting point. We first perform 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 the second step we would perform the same mapping for the source code in the versions before that commit and after the next commit. If the version of the source code at the time of committing the model maps to most model elements, we select that version of the source code. Otherwise, we started a step-wise comparison. If the code in the previous version was best at covering the modeled elements, we performed this step-wise comparison backward through the commit history. Otherwise, we went forward. With each step, we took the next version of the source code (forward or backward) and created a mapping to the model. If the mapping covered more elements of the model compared to the version looked at before, we continue the search. We did that until we found no more improvement and selected the version of the source code that, among the studied ones, offered the greatest coverage of the model.

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

C. Analysis

The analysis was done manually.

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 the 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. Also here a complete manual analysis, considering roles of classes and semantics of modeled relationships was necessary.

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 the 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 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.

D. 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. For three of the five projects studied, the finally 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, 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. 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. Altogether, we studied 466 source code classes with together 1352 attributes and 2634 operations, which had to be matched to 338 classes, attributes, operations, and relationships from the models.

IV. 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.

A. Real Abstraction

Table II summarizes the observed cases of real abstraction.

1) Omissions: The first and, probably, least surprising group of abstractions are the omissions. We find omissions of subsystems, inheritance structures, classes, attributes, attribute types, default values, operations, parameters, parameter names, return types, and relationships.

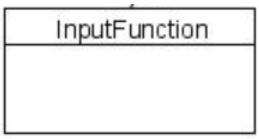
Subsystem omission is a special case, as not all models aim to show an abstraction of the complete system. We found subsystem omission in 3 out of the 5 cases (cases 3, 4, and 5). In two cases, the model focused only on a very specific part and feature of the system and omitted all other system parts. 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 and 5 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. **Inheritance structure omission** can be seen as a special case of **class omission**. However, here the omitted classes are still represented in the model through their superclass. This representation is missing for most classes affected by **class omission**. For example, a case of **inheritance structure omission** can be seen in Figures 1a and 1b. Here the subclasses *Max* and *Min* derived from the superclass *InputFunction* are 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.

TABLE II: Cases of real abstraction.

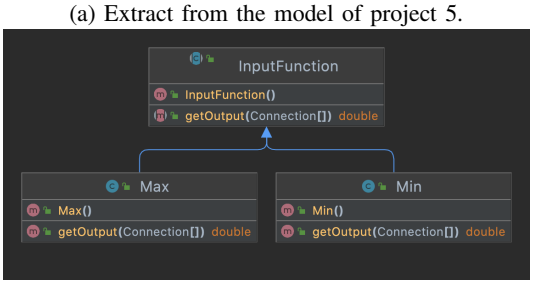
Real Abstraction
Subsystems
Subsystem omission: The model focuses on one or more subsystems of the system only and omits all information about other parts of the code.
Classes
Inheritance structure omission: Inheritance structures in the source code are not shown in the model
Class omission: Classes 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.
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.
Collection type underspecification: Either 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.
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.

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 Figure 2a indicates an *observer* design pattern [35], based on the naming of the class *PetObserver*, its operation *update(): void* of *PetObserver*, and the association between *PetObserver* and the 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 2b illustrates that the operation *getCurrentPet()* of *RaiseMeUp* is

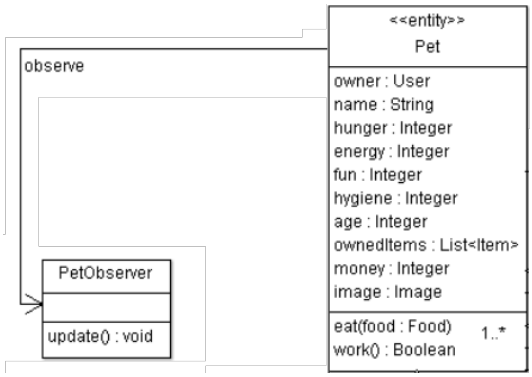


(a) Extract from the model of project 5.



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

Fig. 1: The subclasses and operations are hidden in the model.



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

```
public class PetObserver {
    public PetObserver() {
        /**
         * First, getCurrentPet() of RaiseMeUp invoked
         * by PetObserver.
         * Second, getEnergy() of Pet invoked by RaiseMeUp.
         */
        prevEnergy = RaiseMeUp.getCurrentPet().getEnergy();
    }
}

public class Pet {
    public int getEnergy() {
        return energy;
    }
}
```

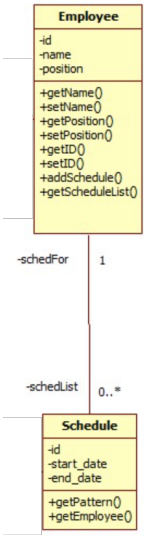
(b) Visualization of corresponding code part from project 2.

Fig. 2: The model shows an association between *Pet* (origin) and *PetObserver* (target). In the source code *PetObserver* only indirectly accesses *Pet* via another class *RaiseMeUp*.

invoked within *PetObserver* and the operation *getEnergy()* of *Pet* is further invoked with the help of *RaiseMeUp*.

3) Other abstractions: We observed two special cases of real abstraction. The first case is the **collection type underspecification**. 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.

The second 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 3a illustrates that in the model of project 4, a bidirectional association is modeled between the classes *Employee* and *Schedule*. In the source code (Figure 3b) the type of the attribute *schedList* (*ArrayList<Schedule>*) indicates that a group of instances of the class *Schedule* is referenced by the class *Employee*, representing a uni-directional relationship. These instances are further initiated within that class (in operation *Employee(Integer id, String nm, String position)*).



(a) Extract from the model of project 4.

```
public class Employee {
    // Complete implementation not shown here.
    // Collection instances of Schedule declared here.
    private ArrayList<Schedule> schedList;
    public Employee(Integer id, String nm,
    String position) {
        // Instances of Schedule created here.
        schedList = new ArrayList<Schedule>();
        // Complete implementation not shown here.
    }

    public void addSchedule(Schedule sch) {
        // add() of Schedule invoked here.
        schedList.add(sch);
    }
}
public class MainClass {
    // Complete implementation not shown here.
    public void doMain() {
        Employee k = new Employee(1,"Marcel Codrea",
        "Trainee");
        // Complete implementation not shown here.
}
}
```

(b) Visualization of corresponding code part from project 4.

Fig. 3: The ownership between *Employee* and *Schedule* from the source code is shown as a simple association in the model.

Observation: Real abstraction cases observed include omissions, summaries, relationship loosening, and collection type underspecification.

B. Disagreements

Table III summarizes the observed cases of disagreement.

1) Pretences: The most common form of disagreement that we observed is the pretence, where the model shows structures that are not present in the source code. We observed such pretence occurrences for inheritance structures, attributes, operations, parameters, and relationships. Figures 4a and 4b show an example of an **inheritance structure pretence** 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

TABLE III: Cases of disagreements

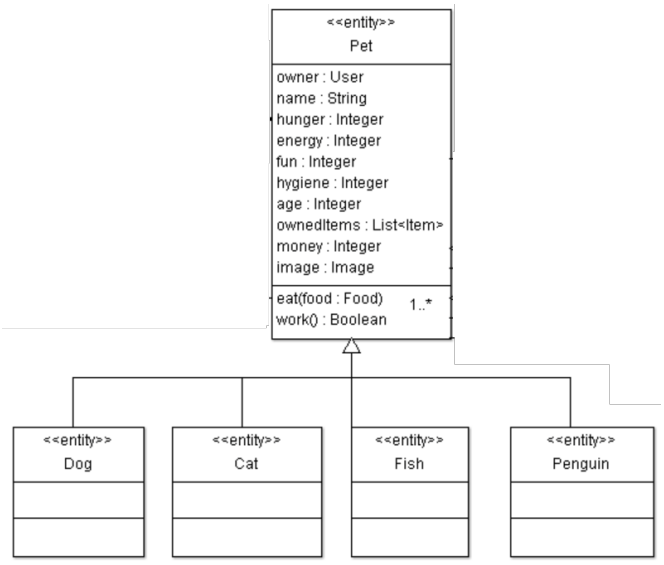
Disagreements
Classes
Inheritance structure pretence: Inheritance structures in the model are not present in the source code
Attributes
Attribute pretence: Attributes shown in the model are not present in the source code
Attribute substitution: Attributes in the model are replaced by different attributes (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 pretence: Operations shown in the model are not present in the source code.
Parameter pretence: Parameters shown in the model are not present in the source code.
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 pretence: Relationships shown in the model are not present in the source code.

the difference between the pet types using an attribute *type*. In the same figures we also see an **operation pretence**, where an operation *eat(food: Food)* is shown in the model, which does not occur in the source code.

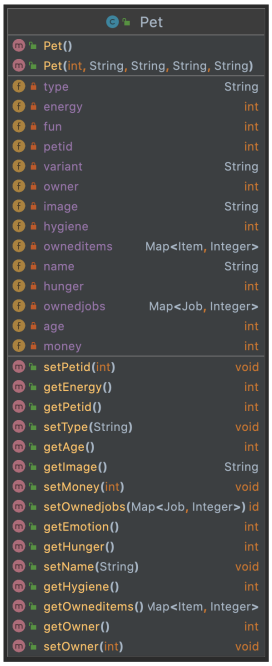
2) Substitution: Substitutions are cases where a structure (most often a type) shown in the model is substituted/substitutes a different structure (type) in the source code. We found such substitutions for attributes, attribute types, parameter types, and return types. For example, the Figures 5a and 5b show 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 datatypes 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 **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*.

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

The first case is **attribute pull up**, where attributes are moved between superclass and subclasses within an inheritance structure. Figures 6a and 6b show 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



(a) Extract from the model of project 2

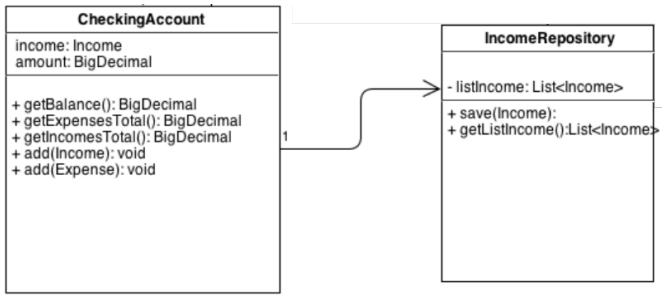


(b) Visualization of corresponding code parts from project 2

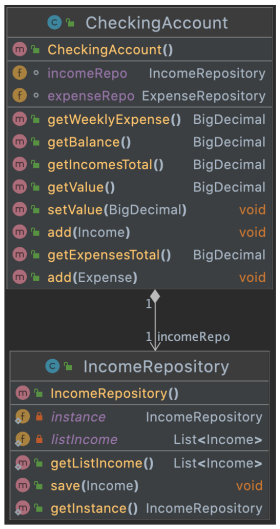
Fig. 4: The model includes an inheritance structure with the superclass *Pet* and an operation *eat(food: Food)*, which are both not present in the source code.

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



(a) Extract from the model of project 3



(b) Visualization of corresponding code parts from project 3

Fig. 5: The model shows an attribute *amount: BigDecimal*, which is substituted by another attribute *expenseRepo: ExpenseRepository* in the source code.

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 Section IV-C) concerning the methods name (*listUser()* vs. *listUsers()*) and a *collection type underspecification* (as explained above).

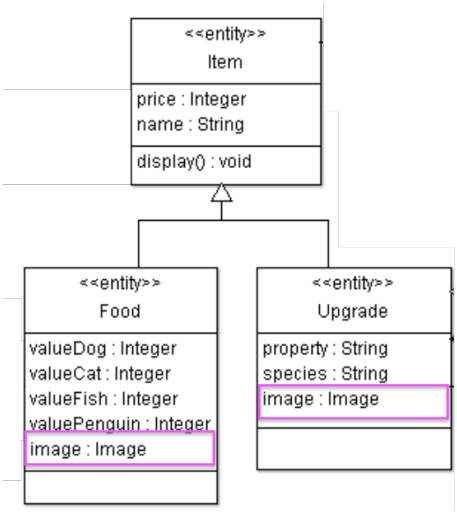
Observation: Disagreements result in the pretence of elements in the model that are not present in the code, substitution of elements, and refactorings between model and code.

C. Inaccuracies

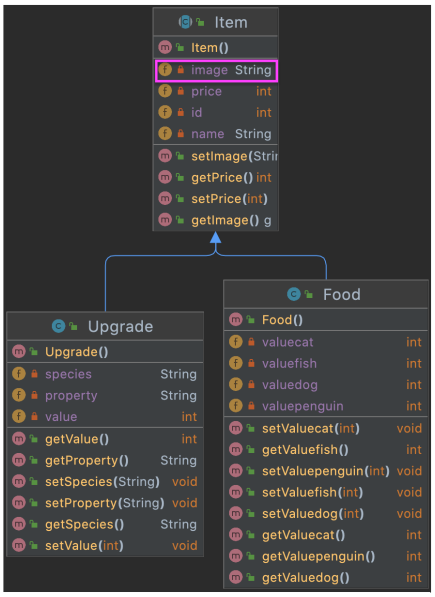
Table IV summarizes identified inaccuracies, which come in two groups: name inaccuracies and type inaccuracies.

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



(a) Extract from the model of project 2.



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

Fig. 6: 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.

class is named *SinglePlayModel* in the model, but *SinglePlayerModel* in the source code. Similarly in project 2, the 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

TABLE IV: 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.
Parameter type inaccuracy: A parameter type from the source code is inaccurately, but recognizably, shown in the model.

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.

2) Type inaccuracies: Types are also sometimes affected by inaccuracies. We found inaccuracies concerning attribute types and parameter types. For example, 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 another case a parameter type *char* was represented as *Char* in the model (class *SinglePlayModel* in project 1).

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

D. Summary of occurrences

Table V 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, and class 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 4 out of 5 projects, future work will need to show whether project 2 is an exception with regards to the variety of disagreements that have been found.

TABLE V: 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	573
Class omission	1, 2, 3, 4, 5	417
Attribute omission	1, 2, 3, 4, 5	245
Relationship omission	1, 2, 3, 4, 5	28
Parameter omission	1, 2, 3, 4	28
Return type omission	1, 2, 3, 4	16
Parameter name omission	1, 2, 3	13
Subsystem omission	3, 4, 5	3
Default value omission	1, 2	7
Collection type underspecification	2, 3	6
Inheritance structure omission	5	25
Attribute type omission	4	15
Operation summary	2	6
Relationship loosening	4	3
Class summary	2	1
Attribute summary	2	1
Relationship summary	2	1
Disagreements		
Operation pretence	2, 4	28
Relationship pretence	2, 5	3
Attribute type substitution	2	6
Operation move	2	5
Inheritance structure pretence	2	4
Attribute substitution	3	2
Attribute pretence	2	2
Parameter pretence	2	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	14
Attribute type inaccuracy	2, 3	16
Parameter name inaccuracy	1, 2	4
Parameter type inaccuracy	1	1

V. Discussion

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 [8] [9]. 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.

Theory building. Our findings partially support findings and design decisions from previous studies on abstraction, partially lead to new findings and add new nuances to existing ones. We indeed found disagreement as a source of differences in four out of five considered projects, which supports previous developer studies which highlight the role of that found deliberate deviations made by developers [4] [5]. Furthermore, our results confirm that classes may be omitted, e.g. if they are view-related or are utility classes, [7] [15]. Yet, even though we found omissions of attributes, only in one case they were systematically omitted. Thus, in contrast to Baltes and Diehl [15], 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. Particularly surprising was that summary elements, as they are proposed by approaches to create abstraction automatically [24] [22], were rarely observed in our subject projects, namely only in one out of five cases.

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., inheritance structure omission vs. inheritance structure pretence, can be considered dual opposites of each other. Such omission-pretence 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 pretence* would not be possible. Nonetheless, this allows 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, e.g., *class pretense* arising as the dual opposite of *class omission*.

VI. 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 [36] and JavaSE7 [37], which were published at almost the same time, 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 instances of generated code. However, our results indicate that we were to some extent successful, due to the many inaccuracies found (which affect all 5 projects, see table V), which one would not expect in cases with generated code (if generated from the studied diagrams).

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. Furthermore, as with all research that is conducted with GitHub repositories, there is a certain risk that the results are not representative of the industrial use of models. Specifically, 3 out of the five projects have a fairly short active period and all projects have below 10 contributors. We also did not assess the quality of the selected projects, which means that we cannot make statements on the representatives regarding 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. Finally, we focused on class diagrams stored in image formats, only. 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.

VII. Conclusion

In this paper, we presented an in-depth study of manual abstraction, focusing on five open-source software projects with their included class models and codebases. We systematically established a collection of mappings between model and code elements, which we analyzed to derive three main types of differences between models and code—real abstraction, disagreements, and inaccuracies—together with lists of cases. Our observed differences shed light on particularly important practices applied during manual abstraction, including *inheritance structure omission*, *collection type underspecification*, and *relationship loosening*.

We foresee the following directions for future work. First, as we discuss in the paper, our results can be used to inform the design and evaluation of improved reverse engineering tools, consistency-checking rules, and manual abstraction guidelines. It would be of particular interest to see how our identified types of manual abstraction relate to abstractions created with the help of large language models. Second, to further improve the reliability of our findings and study the importance of our identified and hypothesized categories of changes, it would be worthwhile to conduct future studies with additional subject systems. Third, future work should investigate manual abstraction in different diagram types beyond class models. Our overarching categories of omission, pretence, and inaccuracies are by no means specific to class diagrams, and it would be interesting to study how they apply to different diagram types.

REFERENCES

[1] A. Dennis, B. H. Wixom, and R. M. Roth, *Systems analysis and design*. John Wiley & Sons, 2008.

[2] B. Vogel-Heuser, A. Fay, I. Schaefer, and M. Tichy, "Evolution of software in automated production systems: Challenges and research directions," *Journal of Systems and Software*, vol. 110, pp. 54–84, 2015.

[3] M. H. Osman and M. R. V. Chaudron, "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)*, 2013, pp. 23–32.

[4] Y.-G. Gueheneuc, "A systematic study of uml class diagram constituents for their abstract and precise recovery," in *11th Asia-Pacific Software Engineering Conference*, 2004, pp. 265–274.

[5] T. Ho-Quang, R. Hebig, G. Robles, M. R. V. Chaudron, and M. A. Fernandez, "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)*, 2017, pp. 203–212.

[6] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[7] M. H. Osman, A. van Zadelhoff, D. R. Stikkolorum, and M. R. V. Chaudron, "Uml class diagram simplification: What is in the developer's mind?" ser. EESSMod '12, 2012.

[8] G. Antoniol, B. Caprile, A. Potrich, and P. Tonella, "Design-code traceability recovery: selecting the basic linkage properties," *Science of Computer Programming*, vol. 40, no. 2, pp. 213–234, 2001, special Issue on Program Comprehension.

[9] D. J. van Opzeeland, C. F. Lange, and M. R. V. Chaudron, "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.

[10] R. Shatnawi and A. Alzu'bi, "A verification of the correspondence between design and implementation quality attributes using a hierarchal quality model," *IAENG International Journal of Computer Science*, vol. 38, no. 3, pp. 225–233, 2011.

[11] R. Hebig, T. H. Quang, M. R. V. Chaudron, G. Robles, and M. A. Fernandez, "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*, ser. MODELS '16, 2016, p. 173–183.

[12] D. Berardi, D. Calvanese, and G. De Giacomo, "Reasoning on uml class diagrams," *Artificial Intelligence*, vol. 168, no. 1, pp. 70–118, 2005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370205000792

[13] Y.-G. Guéhéneuc, "A reverse engineering tool for precise class diagrams," in *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, 2004, pp. 28–41.

[14] OSF. Replication package. [Online]. Available: https://osf.io/p4jdr/?view_only=34b153d3b7704f4a84befd661dc9c6a1

[15] S. Baltes and S. Diehl, "Sketches and diagrams in practice," ser. FSE 2014. Association for Computing Machinery, 2014, p. 530–541.

[16] H. A. Müller, J. H. Jahnke, D. B. Smith, M.-A. Storey, S. R. Tilley, and K. Wong, "Reverse engineering: a roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, 2000, pp. 47–60.

[17] E. Chikofsky and J. Cross, "Reverse engineering and design recovery: a taxonomy," *IEEE Softw.*, vol. 7, no. 1, pp. 13–17, jan 1990.

[18] H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot, "Modisco: A model driven reverse engineering framework," *Information and Software Technology*, vol. 56, no. 8, pp. 1012–1032, 2014.

[19] R. Koschke, "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.

[20] F. Thung, D. Lo, M. H. Osman, and M. R. V. Chaudron, "Condensing class diagrams by analyzing design and network metrics using optimistic classification," in *Proceedings of the 22nd International Conference on Program Comprehension*, ser. ICPC 2014, 2014, p. 110–121.

[21] A. Egyed, "Automated abstraction of class diagrams," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 4, pp. 449–491, 2002.

[22] ——, "Semantic abstraction rules for class diagrams," in *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*. IEEE, 2000, pp. 301–304.

[23] M. Booshehri and P. Luksch, "Condensation of reverse engineered uml diagrams by using the semantic web technologies," in *Proceedings of the International Conference on Information and Knowledge Engineering (IKE)*, 2015, p. 95.

[24] G. Guizzardi, G. Figueiredo, M. M. Hedblom, and G. Poels, "Ontology-based model abstraction," in *2019 13th International Conference on Research Challenges in Information Science (RCIS)*. IEEE, 2019, pp. 1–13.

[25] M. H. Osman, M. R. V. Chaudron, and P. Van Der Putten, "An analysis of machine learning algorithms for condensing reverse engineered class diagrams," in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 140–149.

[26] X. Yang, D. Lo, X. Xia, and J. Sun, "Condensing class diagrams with minimal manual labeling cost," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2016, pp. 22–31.

[27] K.-J. Stol and B. Fitzgerald, "The abc of software engineering research," vol. 27, no. 3, 2018.

[28] G. Robles, T. Ho-Quang, R. Hebig, M. R. V. Chaudron, and M. A. Fernandez, "An extensive dataset of uml models in github," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 519–522.

[29] M. H. Osman, T. Ho-Quang, and M. R. V. Chaudron, "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)*, 2018, pp. 396–399.

[30] Github - zootypers. [Online]. Available: https://github.com/orgs/ZooTypers/repositories

[31] Github - lekogabi/raisemeup. Fork from original. [Online]. Available: https://github.com/WenliZhang1102/RaiseMeUp

[32] Github - antoniorochaoliveira/eapli_pl_2nb. [Online]. Available: https://github.com/AntonioRochaOliveira/EAPLI_PL_2NB

[33] Github - fmacicasan/freedaysintern. [Online]. Available: https://github.com/fmacicasan/FreeDaysIntern

[34] Github - tekosds/neurophchanges. [Online]. Available: https://github.com/tekosds/NeurophChanges

[35] H. Mu and S. Jiang, "Design patterns in software development," in *2011 IEEE 2nd International Conference on Software Engineering and Service Science*, 2011, pp. 322–325.

[36] Object Management Group, Inc., "About the unified modeling language specification version 2.4.1." [Online]. Available: https://www.omg.org/spec/UML/2.4.1

[37] Java Techies Pvt. Ltd. Data types. [Online]. Available: http://jtechies.in/core-java/data-type/java-datatypesd41d.php?