On the Practitioners' Understanding of Coupling Smells – A Grey Literature Based Grounded-Theory Study

Apitchaka Singjai, Georg Simhandl, Uwe Zdun

University of Vienna, Faculty of Computer Science, Research Group Software Architecture, Austria,
`firstname.lastname@univie.ac.at`

Abstract

Context: Code and design smells such as the coupling smells examined in this article are widely studied. Existing empirical studies reveal gaps between the scientific theory and practice, not yet explained by the scientific literature. Only basic coupling smell detection approaches and metrics seem to have been transferred to practice so far.
Objective: This article aims to study the current practitioner's understanding of coupling smells.
Method: Based on grey literature sources containing practitioner views on coupling smells, we performed a Grounded Theory (GT) study. We used UML-based modelling to precisely encode our findings and performed a rigorous analysis of our codes and models.
Results: Our results are defining factors of coupling smells, as well as smell impacts, trade-offs, relationships to other smells, relationships to practices and patterns, and fix options as perceived by practitioners. We further identified gaps in the understanding of coupling smells between science and practice, and derived opportunities and challenges for future scientific work.
Conclusions: Five lessons are presented as opportunities and challenges for future research. Our results can help scientists to get a better understanding of practitioner concerns, and practitioners to get an overview of the current perception of other practitioners on coupling smells.

Keywords: grey literature, grounded theory, design smells, code smells, coupling smells, software design quality, code quality.

1. Introduction

A *Code or Design Smell* is a symptom of poor implementation or design choices that often corresponds to a deeper problem in a software system [1]. The term *Coupling Smell* refers to those kinds of smells that contribute to a specific kind of design problem: excessive coupling between classes. As an example of a coupling smell consider a Method m of a Class A that uses two methods of A itself, but 10 methods of Class B. This method likely suffers from the *Feature Envy* coupling smell because it uses the features of B excessively. Coupling could be substantially reduced, if m is moved to B or – even better – the part of m that uses B could be extracted and then moved to B.

The scientific literature has developed a substantial number of high-quality research studies on code and design smells [2], including metrics to detect code smells [3], smell detection approaches and tools [4, 5, 6, 7, 8], smell fixing approaches and tools [9, 10, 11, 12], taxonomies [13, 14], and a considerable number of empirical studies with human participants [15, 16, 17, 18, 19]. However, the existing empirical studies indicate that there are gaps between the scientific results and practice. For many current scientific approaches, the actual accuracy of smell detection and fixing in complex coupling situations relevant in practice is either low or unclear (see Section 2). We also observed that only relatively simple approaches, such as basic definitions and metrics, seem to have been transferred to practitioner views, approaches, and tools yet. Finally, smells which reside at the boundary between code and design quality, such as *coupling smells*, seem to be more prone to these issues than simplistic code smells such as *Long Method*. These problems have led to this study, which aims to investigate the current practitioner's understanding of coupling smells.

To illustrate the research problem further, let us give a few examples of interesting phenomena not well explained by the current scientific literature on code smell approaches and tools – identified in existing empirical studies on smells: Guo et al. [18] observed that domain-specific tailoring of code smell detection rules and heuristics can greatly improve the human understanding of smells. Furthermore, they found problems in encoding code smells into a tool using metrics from the scientific literature. Mantyla et al. [16] found that the use of smells for code evaluation purposes is hard due to conflicting perceptions of different evaluators, and that metrics and smell evaluations did not correlate. Palomba et al. [17] found that instances of a smell may or may be problematic based on the "intensity" of the problem, and that developer's experience and system's knowledge play an important role in the identification of some code smells. Yamashita and Moonen [19] investigated serious interaction effects between smells. Palomba et al. [17] concluded: "Indeed, there seems to be a gap between theory and practice, i.e., what is believed to be a problem (theory) and what is actually a problem (practice)."

We thus explore the current practitioner's understanding of *coupling smells*. In this article, we describe a *Grounded Theory (GT)* [20, 21] based qualitative study for the above mentioned purpose. We decided to perform a Grey Literature Study (GLS) of acknowledged practitioners' views on *coupling smells*. According to Rainer and Williams [22] there are many benefits of GLS in software engineering research, as they promote the voice of practitioners and provide information on practitioners' contemporary perspectives on important topics relevant to practice and research. In our GT study we used, after initial text-based open coding, formal UML-based modeling for axial and selective coding, instead of the often-used text-based coding process, in order to develop a precisely defined and consistent theory. We have successfully used similar research methods in a couple of prior studies [23, 24, 25, 26]. A secondary contribution of this article is that the description of this method in Section 3 can be used as a definition of this GT variant for later use in other research studies, and the coupling smell study in this context can be seen as a detailed example. We set out to answer the following research questions:

- **RQ1** How are coupling smells understood by practitioners?
  - **RQ1.1** What are the defining factors of coupling smells as perceived by practitioners?
  - **RQ1.2** What are the impacts on and trade-offs to be made with regard to code and design quality when considering coupling smells as perceived by practitioners?
  - **RQ1.3** What are the relations and interactions among coupling smells, to other related smells, and to other software code and design concepts as perceived by practitioners?
  - **RQ1.4** What are the options for fixing coupling smells as perceived by practitioners?
- **RQ2** What are gaps in the understanding of coupling smells between the practitioner's view and the scientific literature?
- **RQ3** What are opportunities and challenges for future scientific work to address the practitioner concerns on coupling smells well?

Our results comprise a set of relevant coupling smells described with a detailed meta-model and a set of definitions describing the defining factors of coupling smells. For each smell, we found many possible liabilities and violations of software design principles that explain their impacts on code and design quality. Our model contains detailed relations among the smells, as well as to existing practices and patterns. Finally, for each smell the model includes detailed options for fixing the smell. We compared our results to previous studies in this field. This revealed interesting gaps, from which we derived five lessons that represent opportunities and challenges for future research.

We believe that our results can help scientists to get a better understanding of practitioner concerns regarding code smells. This can help in better explaining the mentioned empirical results and target future research towards approaches that likely have a high chance of adoption in practice. Our work can also help practitioners to get an overview of the current view of other practitioners on coupling smells.

This article is structured as follows: First, we discuss the related work in Section 2. Next, we explain our research method in Section 3. In Section 4 we present the detailed results of our grounded theory, i.e. the resulting meta-model and the detailed results for the coupling smells. Section 5 discusses the implications for the research questions; this includes an in-depth comparison of our results to the related work. Finally, in Section 6 we discuss threats to validity, and in Section 7 we draw conclusions.

2. Related work

2.1. Research works on coupling and related smells detection and repair

Many studies on coupling smells are about engineering detection and/or smell fixing approaches and tools. Most existing works use metrics or other measures on structural or behavioral features of the code to detect smells. In a recent systematic literature review [2] static source code analysis (such as behavioral, empirical, algorithm-based, methodology-based, and linguistic source code analyses) and dynamic source code analysis (such as dynamic threshold adaptation or genetic algorithms), are found as the primary methods for smell detection. Refactorings, such as those from Fowler's book [1], are often used or suggested as fixes for the smells.

Lanza and Marinescu [3] suggest metrics and thresholds to identify a large number of code smells. The book covers *Feature Envy*, *Data Class* and other smells, as well as situations of intensive coupling underlying many smells discussed in our work. The authors recommend detection strategies, i.e. metrics-based rules, to detect the smells in the code.

JDeodorant [4] is a tool for detecting smells such as *Feature Envy*. Chatzigeoriou and Manakos [27] adopt JDeodorant to investigate three code smells (*Long method*, *Feature Envy*, and *State Checking*) in two open source projects. They recommend to resolve *Feature Envy* by the move method and move field refactorings [1], but the study is limited to the refactoring options offered by the tool. In another work, Tsantalis and Chatzigeorgiou [9] suggest distance metrics as a way to identify move method refactoring opportunities to remove *Feature Envy*. A summary of research related to JDeodorant can be found in [28].

Palomba et al. [5] suggest HIST, a history-based smell detection approach. They compare HIST to JDeodorant for detecting *Feature Envy*, and conclude that the overall F-measure for HIST is 77% and for JDeodorant 68%. For this evaluation, they used a manually produced oracle from 20 open source systems. This work led to the Landfill open data set [29], one of the few manual oracles in the field of code smell.

Distance metrics are often applied in the context of move method refactoring [1], which is one of the most commonly proposed resolution strategies for coupling smells. JMove [10, 11] is based on an approach for move method refactoring recommendation that uses static dependency sets. The precision and recall for JMove [10, 11] were evaluated on synthesized versions of open source systems, in which the authors moved random methods to random classes. It is assumed that a tool should suggest the opposite of the randomly moved methods. JMove's precision ranges from 21% to 32% and its recall ranges from 21% to 60%.

Dipongkor et al. [12] propose another move method refactoring recommendation approach based on the frequency of coupled methods and attributes. Rahman et al. [30] recommend to move methods based on coupling, cohesion, and contextual similarities.

Fontana et al. [31] exploit code smell relations to assess software architecture quality. They identify code smells by recurring anomalies and metrics. Their code smells include among other smells, the *Data Class* and *Message Chain* coupling smells. They evaluate 74 systems to figure out the relationships among the code smells.

Vidal et al. [32] present, prioritize, and evaluate criteria to identify architectural problems using their JSpIRIT tool. 23 versions of four systems are evaluated to reveal smells including *Feature Envy*. For smell detection, they simply use the catalog by Lanza and Marinescu [3] and assume correct identification.

Fard and Mesbah [6] propose automated JavaScript code smells detection in their Jsnose tool which combines a metrics-based approach with static and dynamic analysis. They gather 13 code smells including *Message Chain*. They use rather simple metrics to identify the smells, but most of their smells are simple in nature, too.

Some authors experiment with machine learning approaches for smell detection. Fontana et al. [7] compare 16 machine learning algorithms on four code smells, including *Data Class* and *Feature Envy*. They report a highest accuracy of 96%. However, their training sets are generated by tools using simple metrics and heuristics. Liu et al. [8] suggest deep learning based *Feature Envy* detection based on textual features of the source code. They compare the accuracy of their approach to JDeodorant and JMove. They claim an F1-measure of 18.66% for JDeodorant and 17.27% for JMove, whereas their own approach offers an F1-measure of 52.98%. However, such numbers need to be considered with great care, as the training data is generated based on distance metrics; that is, one of the approaches the authors wanted to improve with their approach is used for training data generation.

2.2. Research works on improving smell detection and repair with additional knowledge

A number of studies try to improve smell detection and repair them with some kind of domain or otherwise specialized additional knowledge. As our study results indicate this as a promising future research direction, we highlight different kinds of related approaches in this section.

With regard to architectural issues, some approaches study the impact of coupling smells on architectural technical debt [31, 32]. A more promising approach might be to study architecture coupling smells directly; e.g. Garcia et al. [33] introduce four architectural smells directly based on architecture concepts and focusing on architectural component interactions. Another even more domain-specific example is the study of Taibi et al. on microservice smells [34] which use architecture structures and specifics of microservices-based architectures to define the smells precisely.

Other approaches focus on other specialized knowledge. Ratiu et al. [35] use historical information to increase the accuracy of automatic detection. They consider the *God Class* and *Data Class* smells. Another history-based approach is suggested by Palomba et al. [5]. Marinescu [36] shows that detection accuracy of the *Data Class* and *Feature Envy* smells can be improved by considering aspects of the studied enterprise applications. Fontana et al. [37] suggest filters to remove false positives, e.g. a filter on test class methods is defined for *Message Chains*, and for *Data Classes* filters such as Serializable Class, Test Class, and Logger Class are defined. Guo et al. [18] present an approach to tailor the detection heuristics to take domain-specific factors into account.

2.3. Systematic studies of coupling and related smells

There are a number of studies on taxonomy and empirical studies of coupling smells. Sabir et al. [2] perform a systematic literature review to study approaches for detection of smells and their evolution in object-oriented and service-oriented systems during 2000-2017. In their coupling category they identify the *Schizophrenic Class*, *Message Chain*, *Middle Man*, *Incomplete Library Class*, *Feature Envy*, *Inappropriate Intimacy*, *Intensive Coupling*, *Extensive Coupling*, and *Unnamed Coupling* smells as coupling smells. The survey identified various static and dynamic source code analyses as the primary methods for smell detection (summarized above).

Mantyla et al. [13] present a subjective taxonomy and provide an initial empirical study on bad smells in code by performing a survey with developers of a small Finnish software company. They categorized 22 smells into 7 categories: Bloaters, Object-Orientation Abusers, Change Preventers, Dispensables, Encapsulators, Couplers, and Others. For the Couplers, they categorize *Feature Envy* and *Inappropriate Intimacy*. Our study of practitioner views include these two smells in this category, too, but reveals that practitioners see many other smells as coupling smells. Marticorena et al. [14] aim to extend the taxonomy of Mantyla et al. [13] with metrics.

Carneiro et al. [15] investigate in an empirical study whether multiple views of a concern have an influence on code smell detection. They investigate the *Feature Envy*, *God Class*, and *Divergent Change* smells. This exploratory study is performed with only 5 participants, so the results are not highly reproducible. The recall and precision for the *Feature Envy* smell were only 0.4 and 0.2, respectively, which confirms points made below related to difficulties related to detecting *Feature Envy* by practitioners.

Mantyla et al. [16] describe the results of a case study in a Finnish software company, where they studied the evaluator effect when subjectively evaluating the existence of smells in code modules. This study is one of the few studies that actually consider subjective impressions of practitioners. The study considers the *Feature Envy* and *Middle Man* smells. Mantyla et al. found that the use of smells for code evaluation purposes is hard due to conflicting perceptions of different evaluators. Secondly, they applied source code metrics for identifying three smells and compared the results to the practitioner evaluations. They concluded that the metrics and smell evaluations did not correlate.

Palomba et al. [17] report about an empirical study on the developers' perception of 12 bad smells. They have shown code entities (that are affected by the smells or not) from three open source projects to developers and master students. The authors found that some smells are generally not perceived by developers as design problems, including *Middle Man* and *Inappropriate Intimacy*. They also found that instances of a smell may or may not represent a problem based on the "intensity" of the problem. Finally, they found that developer's experience and system's knowledge play an important role in the identification of some smells. They claim smells related to possible misuses of OO principles, such as *Feature Envy*, are among those.

Guo et al. [18] present an approach to tailor the metrics and detection rules to take domain-specific factors into account. Input for these domain-specific heuristics is derived from an iterative empirical field study in a software maintenance project. Among others, they consider the *Data Class* smell. They found that the code smell definitions, as proposed by Lanza and Marinescu [3], were judged to be accurate and actionable by a panel of practitioners, but they required tailoring to be applicable in the project. They found that simple tailorings can improve the results. Problems in encoding code smells into a tool called CodeVizard are reported, too, especially that some semantic factors could not be encoded into the tool.

Yamashita and Moonen [19] studied to which extent problems in maintenance projects can be predicted by code smell presence. In a multiple case study six developers working on four different Java systems were observed. Code smells were detected using tools before the study. In-depth examination of quantitative and qualitative data was conducted to determine if the observed problems could be explained by the detected smells (including *Data Class* and *Feature Envy*). Roughly 30% percent of the problems investigated were somehow related to files containing the found code smells. They observed interaction effects amongst code smells, and between code smells and other code characteristics, and these effects led to severe problems during maintenance. They conclude that the role of code smells in the context of the overall system maintainability is relatively minor, and thus they alone cannot predict maintainability issues.

3. Research Method

This article aims to systematically study practitioner knowledge on coupling smells based on a GLS focused on practitioner views on coupling smells.

We performed a GT study in which we used UML-based modeling to precisely encode our findings. GLS is used as the data collection technique for GT. UML-based modelling is introduced with the goal to develop a precisely defined and consistent theory. We have successfully used this and very similar research methods in a couple of prior studies [23, 24, 25, 26]. As it is a rather new combination of research method elements, we explain it and its motivation in this section in detail.

3.1. Grounded Theory

Glaser and Strauss [20] introduced *Grounded Theory (GT)* as research method for discovery of theory from systematically obtained and analyzed data. The method has two unique properties: constant comparison and theoretical sampling [20]. *Constant comparison* means that the method uses an iterative and incremental process of data collection and analysis. GT derives concepts from the data instances and then categories from the derived concepts. Comparisons of data instances with other instances need to be made during data analysis [21], as well as comparisons to concepts and categories. *Theoretical sampling* means that researchers should actively find new data sources driven by the results of the data analysis. Researchers need to be theoretically sensitive [21]; that means, they should think effectively about what types of data need to be collected and which aspects of the data already collected are most important for the theory [38].

In this article, we roughly follow the GT data analysis and coding steps by Corbin and Strauss [21] who designed a highly systematic and rigorous coding structure to create (rather than to discover) a rigorous theory which closely corresponds to the data [39]. They use three main coding steps:

- *Open coding* is performed first after data collection. It aims to examine the data and identify discrete elements in the data. Open coding identifies concepts in the data.
- During *axial coding* the researchers identify categories in the concepts, e.g. by identifying concepts that reappear in the data, synonymous concepts, related concepts, etc.
- *Selective coding* is about carving out main ideas of story lines of the theory, i.e. understanding the big picture by reflecting on the data and analysis results. An important step is the reduction of concept and categories.

GT ends when *theoretical saturation* is reached. This occurs when no new concepts emerge from new data sources anymore and the theory has been thoroughly validated with the collected data [38].

3.2. Grey Literature Study

Grey literature in software engineering can be defined as "any material about software engineering that is not formally peer-reviewed nor formally published [40]." According to Rainer and Williams [22] there are many benefits in using grey literature sources in software engineering research, including that "they provide information on practitioners' contemporary perspectives on important topics relevant to practice and to research" and "promote the voice of practitioners [22]." As our research questions require practitioner views, we decided to focus on grey literature sources stemming from practitioners and mixed groups (practitioners and others). According to Garousi et al. [41] such grey literature sources includes blog posts, presentations, Wiki articles, technical reports, audio-video material, practitioner book content, and many other kinds of sources. Garousi et al. [40] also stated that "grey literature sources can be classified according to the two dimensions: expertise and outlet control." In our work we are interested in (technical) practitioner knowledge targeted at other practitioners. We thus reviewed each source in the author team and only if all authors agreed that the content contains knowledge by acknowledged practitioners we included the source. We excluded sources that seemed to sell a product or a service. As we intended to use grey literature sources in a GT study, the actual search process was guided by GT's theoretical sampling concept and stopped when theoretical saturation emerged.

In social science, very often interviews are used as data sources for GT, but it is clearly described in the literature that GT can be used with any kind of data [20, 21]. Given that we are not studying some general topic which more or less any software developer can speak about without preparation, and as expertise for coupling code smells is rather rare, searching for interview candidates could have led to substantial bias in the selection process. These are the main reasons for using grey literature instead of interviews. Rainer and Williams [22] summarize some other reasons that have been important for this decision, namely: grey literature has compensated for the unavailability of other sources of evidence. We were able to access harder-to-access practitioners, and we could gather information for the research in a non-invasive way. We were able to scale-up the research to larger number of samples, and complement and triangulate them with, other sources of data. Finally, it was easier to provide an audit trail of the research, and thus enable repeatability of the study through public access to original data[1].

Using a search engine to find grey literature results in various types of data [42]. One major concern about search engines in such research is their search algorithm because the results are dependent on the user [43]. To avoid personal bias in the research, our initial search keywords are taken from "A Taxonomy for Bad Code Smells"[2] Thus we have initially searched for the following keywords: "Code Smells", ("Couplers" or "Coupling") and "Smells", "Feature Envy", "Inappropriate Intimacy", "Message Chains", and "Middle Man." We have opted against a Multivocal Literature Review (MLR) where scientific sources are included, as our research goals focus on exploring practitioner views specifically. Our initial data are from the keywords above. After the coding process, further data sources emerged in terms of new keywords or new referenced sources. For example, we added synonyms such as "Object Orgy" or new coupling smell candidates such as "Data Class" after they appeared the first time. The data collection process is repeatable with scanning and skimming techniques. During open and axial coding we studied each included source line by line in-depth during open coding – for most sources in many iterations. We chose this method over manually browsing selected grey literature initially because it is replicable [45]. In our open access appendix, each step in the open, axial, and selective coding, including the emergence of new concepts, is documented. This comes along with traces which parts of the sources led to which codes in all coding stages.

3.3. Methodology overview

Our application of the research method happened in many iterations. That is, we searched for one or a few new knowledge sources, applied the GT coding process (modified with UML-based modeling) to identify candidate categories, and compared with the so-far-designed model continuously. We improved this model incrementally. A crucial question in GT is when to stop this process; here, theoretical satura-

[1]We provide all open and axial coding files, derived coded models in Python, and generated models (in UML, Markdown, and Latex) as a replication package for download for the time of the review: `https://ucloud.univie.ac.at/index.php/s/sR0kCyGfPpWDDEl` Once published, we will make the replication package available in a long-term open access archive such as Zenodo to enable reproducibility of our results.

[2]See `http://mikamantyla.eu/BadCodeSmellsTaxonomy.html` which is part of a scholarly article [44].

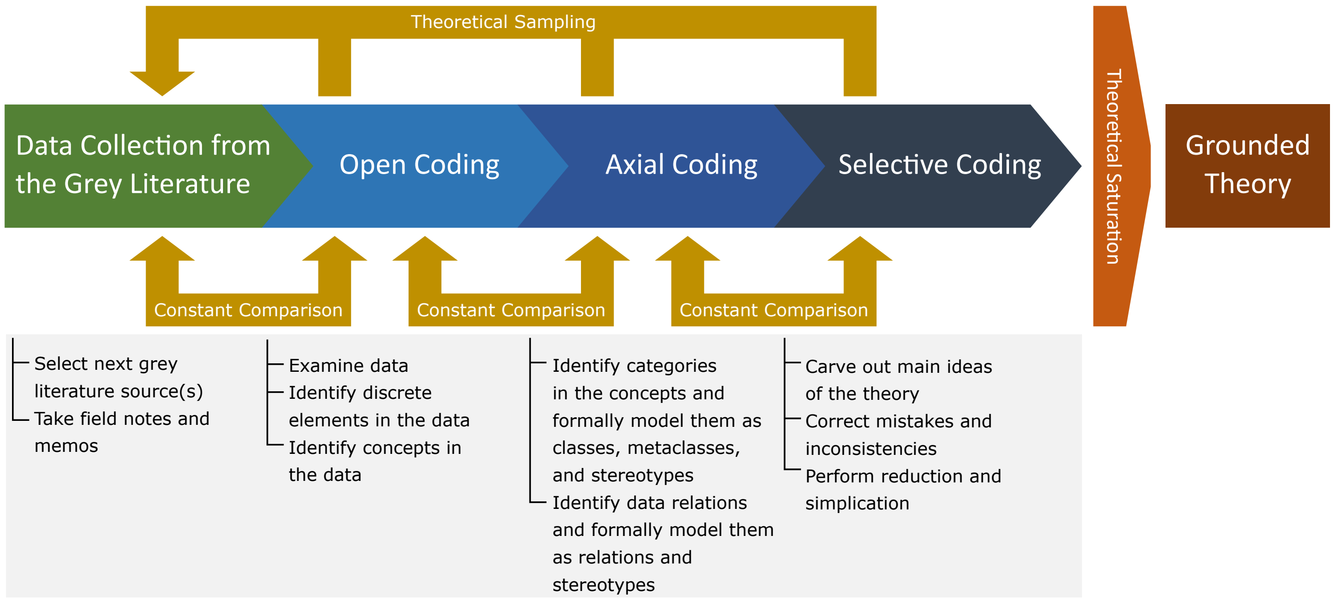

Theoretical Sampling
Data Collection from the Grey Literature
Open Coding
Axial Coding
Selective Coding
Theoretical Saturation
Grounded Theory
Constant Comparison
Constant Comparison
Constant Comparison
Select next grey literature source(s)
Take field notes and memos
Examine data
Identify discrete elements in the data
Identify concepts in the data
Identify categories in the concepts and formally model them as classes, metaclasses, and stereotypes
Identify data relations and formally model them as relations and stereotypes
Carve out main ideas of the theory
Correct mistakes and inconsistencies
Perform reduction and simplication

Figure 1: Research Method Steps

tion [38], explained in Section 3.1, has attained widespread acceptance in qualitative research: We stopped our analysis when 10-15 additional knowledge sources did not add anything new to our understanding of the research topic. As a result of this very conservative operationalization of theoretical saturation, we studied a rather large number of knowledge sources in depth (48 in total, summarized in Table 1) , whereas most qualitative research often saturates with a much lower number of knowledge sources. Our search was based on our own experience and links between the sources. We also used major search engines (e.g., Google, Bing) and topic portals (e.g., InfoQ, DZone) to find relevant practitioner texts. We included knowledge sources, if they were advanced practitioner reports on their experiences or knowledge about coupling smells. We checked that the source texts fulfilled a certain minimum quality level, as described above.

Figure 1 illustrates the GT research method steps as explained in Section 3.1 in the upper half of the figure. In the grey box beneath the steps, we detail our specific research methodology steps. In particular, it can be seen that in the data collection step, we used grey literature as input for the data collection. The open coding step has been performed very closely to how it would be performed without UML-based modeling. In the axial coding step, we have then formalized the found concepts and categorized them using classes, meta-classes, and stereotypes, as well as their formal relations and relation stereotypes. Constantly, during the selective coding step, we have compared the results of the axial coding with the big picture to carve out main ideas of the theory, to correct mistakes and inconsistencies, and to perform reduction and simplification. Those steps have been repeated for each data source.

For modelling we used our existing CodeableModels tool[3], a Python implementation for precisely specifying meta-models, models, and model instances in code with an intuitive and lightweight interface. Based on CodeableModels, we specified a meta-model and models for our study, extending both as needed. In addition, we realized automated constraint checkers and PlantUML code generators to generate graphical visualizations of all meta-models and models, as well as a full textual model output generation in Markdown and Latex.

[3]`https://github.com/uzdun/CodeableModels`

ID	Title	Archive URL	Author Type	Source Type	Example	Source Code
S1	What is the difference between Inappropriate Intimacy and Feature Envy?	https://bit.ly/3bFyKQT	Practitioner	Discussion Forum Post	False	False
S2	Code Smell	https://bit.ly/2S6Ca7K	Mixed	General Audience Article	False	False
S3	Code Smells	https://bit.ly/354s905	Practitioner	Practitioner Audience Article	False	False
S4	Code Smells	https://bit.ly/2VVaELq	Practitioner	Practitioner Audience Article	False	False
S5	Smells to Refactorings Cheatsheet	https://bit.ly/2VBtqbE	Practitioner	Practitioner Audience Article	False	False
S6	Feature Envy	https://bit.ly/2VApxE4	Practitioner	Tool Documentation	True	True
S7	Data Class and Feature Envy	https://bit.ly/2yO90mP	Practitioner	Practitioner Audience Article	False	False
S8	Code Smell: Data Class	https://bit.ly/2znerJN	Practitioner	Practitioner Audience Article	True	False
S9	Feature Envy Smell	https://bit.ly/2KCO3hi	Practitioner	Practitioner Audience Article	False	False
S10	Spotting Feature Envy and Refactoring	https://bit.ly/355ugAz	Practitioner	Practitioner Audience Article	True	True
S11	Resolving Feature Envy in the Domain	https://bit.ly/3cLhkm1	Practitioner	Practitioner Audience Article	True	True
S12	Object Orgy	https://bit.ly/2Y4XM8k	Practitioner	Practitioner Audience Article	False	False
S13	Object Orgy	https://bit.ly/3bD3u5j	Practitioner	Practitioner Audience Article	False	False
S14	Code Smells	https://bit.ly/2yFH6cO	Mixed	General Audience Article	True	True
S15	Refactoring Couplers	https://bit.ly/354O72Z	Practitioner	Practitioner Audience Article	True	True
S16	When to Start Refactoring Code and When to Stop	https://bit.ly/2xdpc0M	Practitioner	Practitioner Audience Article	True	False
S17	Improving Application Design with a Rich Domain Model	https://bit.ly/2VAqT1C	Practitioner	Slides	True	True
S18	Code Smell: Feature Envy or Data Envy	https://bit.ly/2W0Q5Nx	Practitioner	Practitioner Audience Article	True	True
S19	Practical PHP Refactoring: Remove Middle Man	https://bit.ly/2S78tTY	Practitioner	Practitioner Audience Article	True	True
S20	Rich Domain Model with DDD/TDD Reviewed	https://bit.ly/2VDK36P	Practitioner	Practitioner Audience Article	True	False
S21	Feature Envy in Component Design	https://bit.ly/2VD99mc	Practitioner	Practitioner Audience Article	True	True
S22	Write clean code and get rid of code smells with real life examples	https://bit.ly/3aCg57r	Practitioner	Practitioner Audience Article	True	True
S23	Middle Man Code Smell Resolution with examples	https://bit.ly/3aE1lVA	Practitioner	Practitioner Audience Article	True	True
S24	Bad smell in Code Inappropriate Intimacy	https://bit.ly/3bF2Yng	Practitioner	Practitioner Audience Article	True	True
S25	Inappropriate Intimacy Code Smell Resolution	https://bit.ly/2Y7fCav	Practitioner	Practitioner Audience Article	True	True
S26	Patterns in Practice Cohesion And Coupling	https://bit.ly/3bF3GRs	Practitioner	Practitioner Audience Article	True	True
S27	Feature envy	https://bit.ly/2xQZuQ2	Practitioner	Practitioner Audience Article	True	True
S28	Refactoring a Feature Envy Code	https://bit.ly/2KLOqqh	Practitioner	Practitioner Audience Article	True	True
S29	Feature Envy - Code Smell	https://bit.ly/353QLpP	Practitioner	Practitioner Audience Article	True	True
S30	Class: Reek::Smells::FeatureEnvy	https://bit.ly/2KzyC9Q	Practitioner	Tool Documentation	True	True
S31	Feature Envy Code Smell Resolution with examples	https://bit.ly/2VDr7Vz	Practitioner	Practitioner Audience Article	True	True
S32	Why are data classes considered a code smell?	https://bit.ly/2VQ4vAa	Practitioner	Discussion Forum Post	True	False
S33	Coding Best Practices: Clean Code, Refactoring and TDD	https://bit.ly/2W0Dlqy	Practitioner	Practitioner Audience Article	False	False
S34	Data Class - Is It Really A Smell?	https://bit.ly/2KIuggR	Practitioner	Discussion Forum Post	False	False
S35	Refactoring: Code Smells	https://bit.ly/3bC44jw	Practitioner	Slides	False	False
S36	Identifying Code Smells In Java	https://bit.ly/2VQQAtE	Practitioner	Practitioner Audience Article	True	True
S37	How to identify a Data Class using NDepend	https://bit.ly/2S5q6n2	Practitioner	Tool Documentation	True	True
S38	Sharpen your sense of code smell	https://bit.ly/3cNOotY	Practitioner	Practitioner Audience Article	False	False
S39	How to find the code smell AND do not let it go bad, part 2	https://bit.ly/2VWobT6	Practitioner	Practitioner Audience Article	True	False
S40	Everything you need to know about Code Smells	https://bit.ly/2znK9Xj	Practitioner	Practitioner Audience Article	False	False
S41	How to identify a Data Class using NDepend	https://bit.ly/2Ygi26W	Practitioner	Tool Documentation	True	True
S42	thoughts on feature envy	https://bit.ly/3aGIg5a	Practitioner	Practitioner Audience Article	False	False
S43	Feature envy	https://bit.ly/2YfTA5D	Practitioner	Practitioner Audience Article	True	True
S44	Inappropriate Intimacy	https://bit.ly/3f1i3Bv	Practitioner	Practitioner Audience Article	False	False
S45	Feature Envy	https://bit.ly/2KItgJB	Practitioner	Tool Documentation	True	False
S46	Exploring Smelly Code	https://bit.ly/2YigCJ8	Practitioner	Practitioner Audience Article	True	True
S47	Code Smell: Feature Envy	https://bit.ly/35k6UY7	Practitioner	Practitioner Audience Article	True	True
S48	Inappropriate Intimacy	https://bit.ly/3d16DMe	Practitioner	Practitioner Audience Article	False	False

Table 1: Overview of studied knowledge sources

4. Grounded theory on coupling code smells

4.1. Meta-model

The meta-model of our found theory defines the classification (or categories) of model elements, as well as their relations, necessary to model the design space of coupling code smells fully. Of course, many alternative ways to model that design space are possible, such as different naming of model elements or using more meta-classes instead of stereotypes. Nonetheless, the meta-model reveals interesting insights of which conceptual elements that are present in practitioner discussions of coupling code smells.

We first present the basic meta-classes derived in the GT study in Figure 2[4]. The basis of our study are *Sources* which are texts from which we extracted the GT *Codes*. *Sources* are specific kinds of *References* used as knowledge sources in the study; other references are used to model links to study-external sources (such as a book or scientific paper cited by the practitioners). We describe each *Reference* using a couple of meta-data attributes: the title, a URL, an archive URL linking to the Web archive version of the source (to enable reproducibility of our results), and an optional additional bibliographic reference is used to identify the source. The author type is an enumeration with possible values *Practitioner*, *Academic*, *Mixed*, and *Unknown*; in our study we only included sources with *Practitioner* and *Mixed* author types. The type of the source describes which kind of grey literature is used. In our study we included *Discussion Forum Post*, *General Audience Article*, *Practitioner Audience Article*, *Tool Documentation*, *Blog Post*, and *Slides* as source types. Further, we indicate whether an example was provided, and whether the example, if present, includes source code.

Each *Code* is described with a name and an optional description. The core code is the *Smell*, which can be linked to other smells. The smell has a *Definition*, which for coupling smells is in detail specified with the origin and target types in the coupling relation. These are two additional enumerations. In our study we have found evidence for *method*, *class*, *classes*, or *code fragment* as origins of the coupling, and *a class's methods*, *a class's public features*, *a class's features*, *a class's private features*, *a class's implementation details*, *classes*, *a class's data*, *methods of classes*, *private features of classes*, and *clients* are possible targets of the coupling. Moreover, a smell can have *Fixes*. It has a number of *Liabilities* associated which make it a bad smell in code or design that should be resolved. Similar are violations to *Principles* the smell can cause. The *Smell*, as well as its *Liabilities* and its *Principle* violations can cause *Technical Debt*.

The *Association* meta-class of the inter-smell relationship is in our model extended by the stereotype *Inter-Smell Relation Type*. We have found evidence for the kinds of inter-smell relations depicted at the top of Figure 3. *Smells* can be associated to certain (best or common) *Practices* in our field. Many such best practices are described in the literature as *Patterns*. We have found evidence for the kinds of smell-practice relations depicted at the bottom of Figure 3.

4.2. Coupling smells and their relationships

Figure 4 gives an overview of the coupling smells and their relations. As can be seen we identified *Feature Envy*, *Inappropriate Intimacy*, *Data Class*, *Indecent Exposure*, *Message Chain*, and *Middle Man* as immediate sub-classes (is-a relation) of coupling smell. Those major coupling smells will each be discussed in their own subsection below. *Data Envy* is a special kind of *Feature Envy* and will be discussed together with it below. Those are only included in our study if there was clear evidence that a coupling smell has some kind of strong relation to the other smell. We only referenced those other smells, but did

[4]Please note that the UML figures used in this article are directly generated from our coded models in our tool using PlantUML. The figures are only touched to optimize the layout for the article slightly.

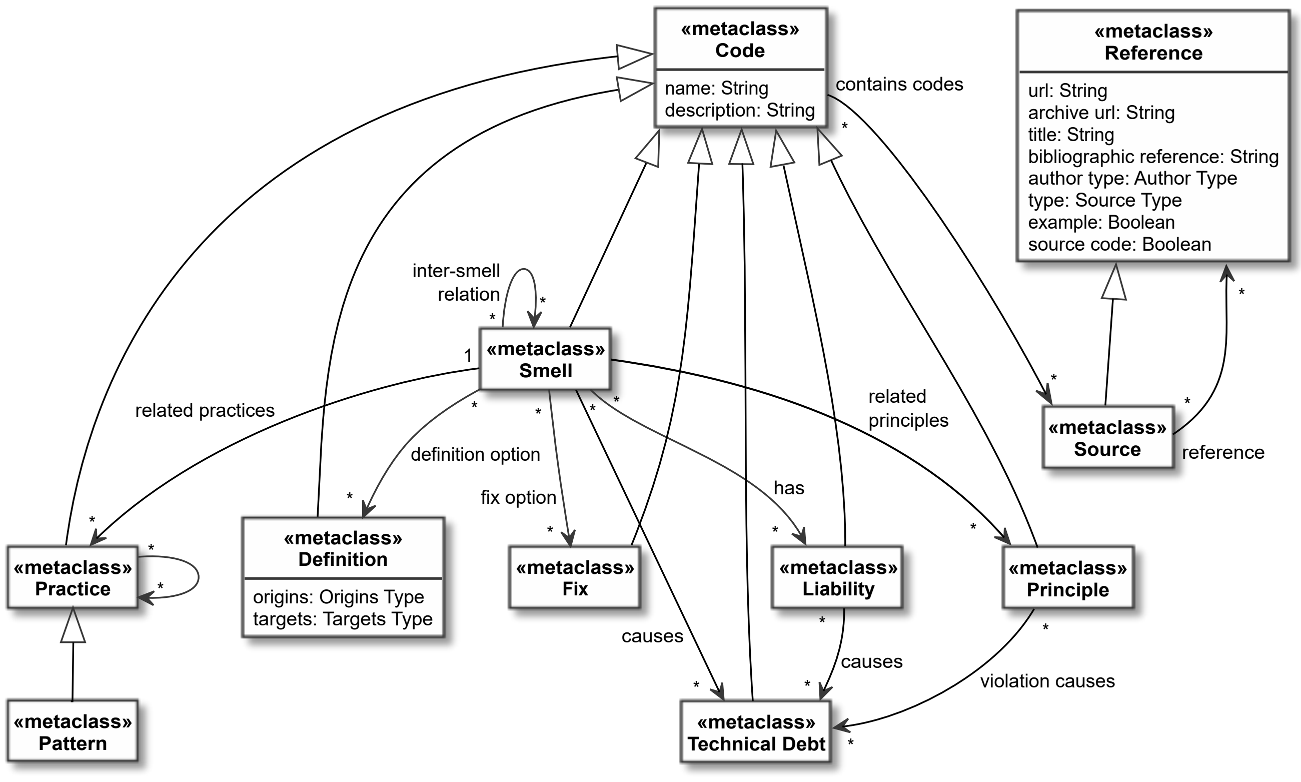
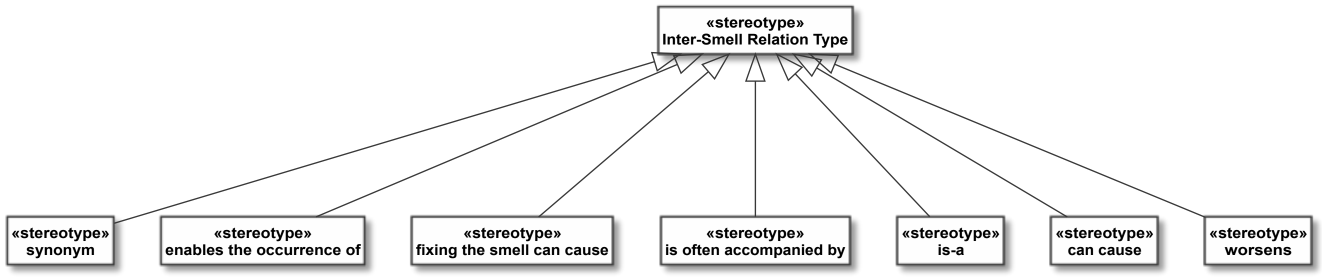


Figure 2: Meta-model main meta-classes


«stereotype» Inter-Smell Relation Type
«stereotype» synonym
«stereotype» enables the occurrence of
«stereotype» fixing the smell can cause
«stereotype» is often accompanied by
«stereotype» is-a
«stereotype» can cause
«stereotype» worsens

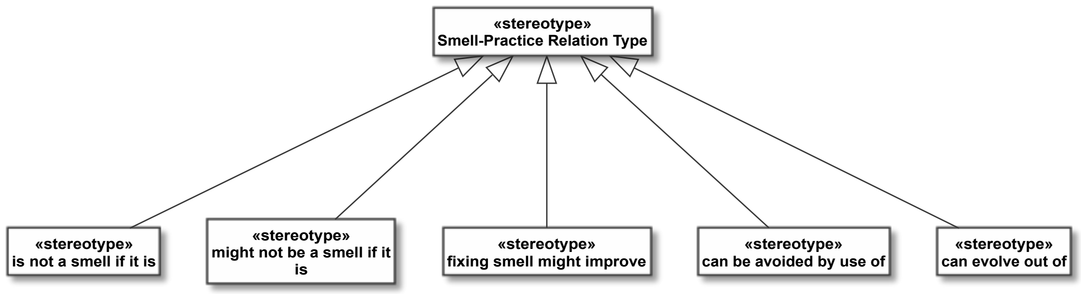

«stereotype» Smell-Practice Relation Type
«stereotype» is not a smell if it is
«stereotype» might not be a smell if it is
«stereotype» fixing smell might improve
«stereotype» can be avoided by use of
«stereotype» can evolve out of

Figure 3: Meta-model stereotypes

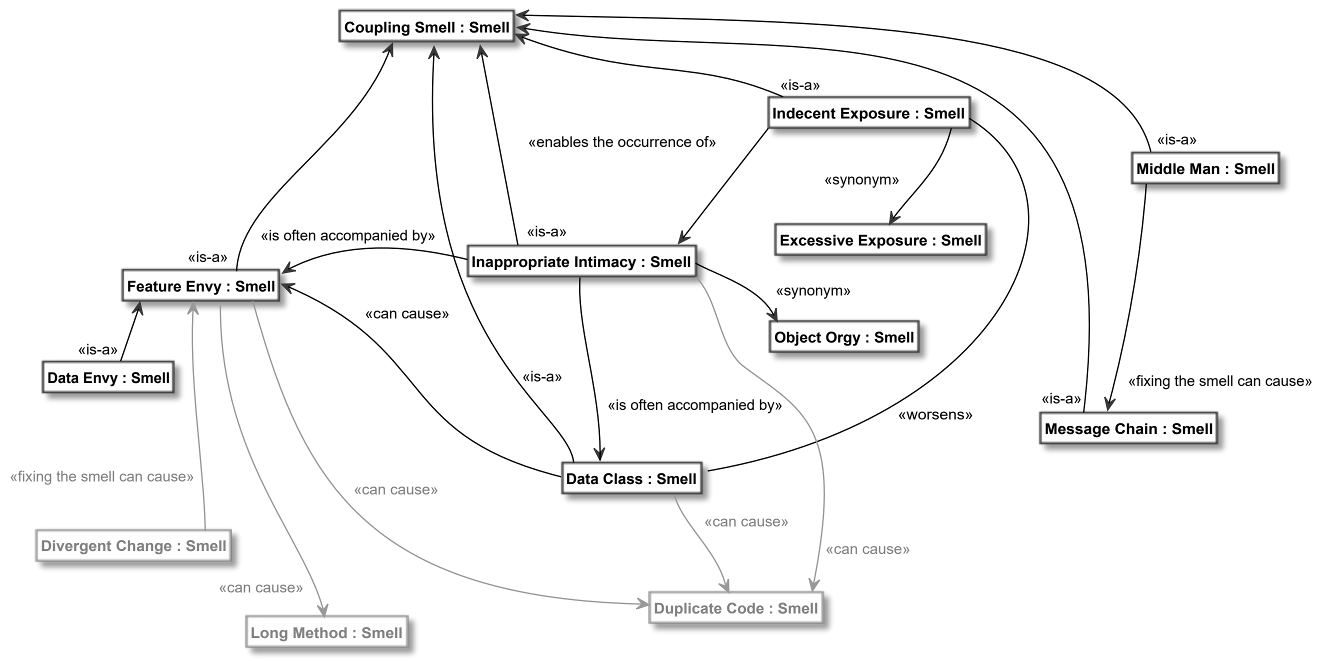


Figure 4: Coupling smells and their relations (non-coupling smells are rendered in grey)

not study them in detail. The coupling smells and their inter-smell relationships are discussed separately per smell in the following sections.

Table 2 presents all smells and color coding[5] in the second row indicates how often the smell appears in our total of 48 sources. All relations are listed twice for their source and target smell, and the different color coding in the fourth row indicates how often the relation appears in the total number of sources of the smell (from the second row).

4.3. *Feature Envy*

4.3.1. *Definitions*

Feature Envy is a very commonly discussed coupling smell. With 33 evidences for the smell it occurs most frequently in our sources. With a total of 9 different definition variants shown in Table 3, it also has the largest number of possible interpretations. There are two common relations between coupling origin (i.e. the place where *Feature Envy* can be spotted) and the coupling targets (i.e. the places the origin is envious of): 1. the origin makes excessive use of the target; 2. the origin uses specific target features more often than its own. We classified the definitions in the table into two main variants: Definition Variant 1 is the broader definition, whereas Variant 2 is often used in sources (or derived from other original source) where automated detection or similar operationalization is a goal or at least considered to be a goal. That is, "excessive use" is less specific and thus hard to judge automatically, whereas "more features of a specific kind" can be easily counted and compared. Note that Variant 2 is problematic in the sense that it is easy to construct examples which a human reviewer usually would not accept as *Feature Envy*, but which would conform to the definition.

The sources are also split among different options where *Feature Envy* occurs (i.e., the origin): Most see a single method (or operation in non-object-oriented languages) as the typical *Feature Envy* origin,

[5]In the tables we use the following color coding to visualize the frequency of the evidences: 🟦 < 5%, 🟦 < 10%, 🟦 < 20%, 🟦 < 35%, 🟦 < 50%, 🟦 < 70%, 🟦 >= 70%

Smell	**All Smell Evidences**	**Relations**	**Relation Evidences**
Feature Envy	33	Data Envy *is-a* Feature Envy	S9, S18
		Feature Envy *can cause* Duplicate Code	S4, S10, S42, S46
		Inappropriate Intimacy *is often accompanied by* Feature Envy	S15
		Feature Envy *can cause* Long Method	S17
		Data Class *can cause* Feature Envy	S7, S17, S28, S39
		Divergent Change *fixing the smell can cause* Feature Envy	S14, S16
Inappropriate Intimacy	21	Inappropriate Intimacy *synonym* Object Orgy	S2, S12, S13
		Indecent Exposure *enables the occurrence of* Inappropriate Intimacy	S3, S5, S15
		Inappropriate Intimacy *is often accompanied by* Data Class	S12
		Inappropriate Intimacy *can cause* Duplicate Code	S15
		Inappropriate Intimacy *is often accompanied by* Feature Envy	S15
Data Class	16	Inappropriate Intimacy *is often accompanied by* Data Class	S12
		Data Class *can cause* Duplicate Code	S4, S36
		Indecent Exposure *worsens* Data Class	S8
		Data Class *can cause* Feature Envy	S7, S17, S28, S39
Indecent Exposure	6	Indecent Exposure *enables the occurrence of* Inappropriate Intimacy	S3, S5, S15
		Indecent Exposure *worsens* Data Class	S8
		Indecent Exposure *synonym* Excessive Exposure	S15
Message Chain	12	Message Chain *fixing the smell can cause* Middle Man	S4, S39
Middle Man	13	Message Chain *fixing the smell can cause* Middle Man	S4, S39
Data Envy	3	Data Envy *is-a* Feature Envy	S9, S18
Object Orgy	3	Inappropriate Intimacy *synonym* Object Orgy	S2, S12, S13
Excessive Exposure	1	Indecent Exposure *synonym* Excessive Exposure	S15
Duplicate Code	4	Feature Envy *can cause* Duplicate Code	S4, S10, S42, S46
		Data Class *can cause* Duplicate Code	S4, S36
		Inappropriate Intimacy *can cause* Duplicate Code	S15
Long Method	1	Feature Envy *can cause* Long Method	S17
Divergent Change	2	Divergent Change *fixing the smell can cause* Feature Envy	S14, S16

Table 2: Overview of coupling smells, their relations, and evidences

but some see a whole class or a code fragment (e.g. in a method) as places to spot *Feature Envy*. The class interpretation can be difficult to judge, as some methods might be envious while many others are not. Also, it is easy to create long methods with code fragments in it that are envious, where the rest of the long method is not. Thus the code fragment interpretation sounds appealing in that sense, but would also make judgments whether it is *Feature Envy* or not more difficult.

For the coupling target, the sources agree that it is a class, but they see envy occur with different kinds of features, namely all the class's features, its public features, its data, or its methods. Methods and all features are the most frequent options. Note that if all data is accessed via method (i.e., no *Indecent Exposure* of data occurs), the variety in targets is rather limited.

The following code shows a simple *Feature Envy* example[6]. *getTotalPrice()* is a method envious of the features in *Item* according to both definitions, as it uses 4 methods of *Item* and 0 methods of *Basket*.

```
class Item { .. }
  class Basket {
  // ..
  float getTotalPrice(Item i) {
    float price = i.getPrice() + i.getTax();
    if (i.isOnSale())
      price = price - i.getSaleDiscount() * price;
    return price;
  }
}
```

The simple example can also be used to explain the issues with the Variant 2 definition ("more features of a specific kind") and code fragment vs. method as origin easily. Consider we add a statement that uses 4 *Basket* features to the method: Then the code fragment in the first three lines is just as envious as before, but now it is not *Feature Envy* according to the Variant 2 definition anymore (assuming *method* as origin).

```
float getTotalPrice(Item i) {
  float price = i.getPrice() + i.getTax();
    if (i.isOnSale())
      price = price - i.getSaleDiscount() * price;
      if (basketContainsGroupDiscountItems() || isLargeBasketDiscount() || isPriceDiscount())
        price = price - basketDiscounts();
    return price;
  }
}
```

The discussion in this section might suggest that there are quite different views on *Feature Envy* in the practitioner literature, but when reflecting on the practitioner texts more deeply, this is not really the case. In cases where the practitioners do not just take over a definition from another source, but discuss it in more detail, it becomes clear that even if they use Definition Variant 2, they rather interpret it loosely. In contrast, a strict interpretation has the great risk of producing a lot of false positives or negatives in automated tools.

4.3.2. Relationships to other smells

As shown in Figure 4, sometimes *Feature Envy* is discussed under the term *Data Envy*, which thus has an is-a relation to *Feature Envy*. It occurred relatively seldom (3 sources) and was defined using the same two definitions types as observed for *Feature Envy*.

Feature Envy can be caused by *Data Classes*. That is, other classes might be envious to the data access features offered by the *Data Class*. It is often accompanied by *Inappropriate Intimacy*, e.g. if the envied

[6]Note that we give a few simplistic example to illustrate some of the smells. While they conform to the smell definitions, they alone might not be seen as harmful by many practitioners. According to our results smells become harmful, when their "intensity" is high enough.

Definition	Coupling origin(s)	Coupling target(s)	Definition Evidences
Feature Envy			Number of evidences containing a definition: 35
Method uses class excessively	a method	a class's methods	S1, S3, S5, S14, S15, S22, S26, S31, S33, S47
Method uses features of another class more often than its own features	a method	a class's features	S6, S7, S9, S10, S11, S14, S16, S17, S21, S27, S35, S36, S40, S41, S43, S45, S46
Class uses features of another class more often than its own features	a class	a class's features	S31
Method uses public features of another class more often than its own features	a method	a class's public features	S1
Class uses class excessively	a class	a class's methods	S2, S14, S18, S31
Method uses more data of other class more than its own class's data	a method	a class's data	S4, S29, S31, S38, S39
Class uses more data of other class more than its own data	a class	a class's data	S20
Class uses data of another class excessively	a class	a class's data	S28
Code fragment uses features of another class more often than its own features	a code fragment	a class's features	S30
Inappropriate Intimacy			Number of evidences containing a definition: 20
Class uses other class's private features excessively	a class	a class's private features	S1, S24, S35
Class uses other class's implementation details	a class	a class's implementation details	S1, S2, S4, S12, S13, S22, S25, S26, S38, S40
Two or more classes depend on each other excessively	classes	classes	S3, S15, S33, S36
Two or more classes depend on each other's private features excessively	classes	private features of classes	S5, S14
Method uses other class's implementation details	a method	a class's implementation details	S38, S44, S48
Data Class			Number of evidences containing a definition: 14
Class only has data, getters, and setters	a class	classes	S3, S4, S5, S7, S8, S12, S14, S16, S17, S32, S33, S35, S36, S37
Indecent Exposure			Number of evidences containing a definition: 5
Class exposes internal details	a class	clients	S3, S5, S15, S33, S38
Message Chain			Number of evidences containing a definition: 12
Long sequence of method calls	a class	methods of classes	S3, S4, S5, S14, S15, S16, S22, S33, S35, S36, S39, S40
Middle Man			Number of evidences containing a definition: 14
Class only delegates to other classes	a class	classes	S3, S4, S5, S15, S16, S22, S23, S39
Class is merely a wrapper over the other class	a class	classes	S3, S33
Class delegates the majority of the work to other classes	a class	classes	S14, S33, S35, S40
Method is merely a wrapper hiding delegates	a method	classes	S19, S38

Table 3: Definitions of coupling smells and evidences. Color coding indicates how many evidences of a smell containing a definition contain the specific definition.

class also has an intimate relation to the envious class or if the envious class uses implementation details of the envied class.

Patterns such as *Strategy* and *Visitor* are used to fix the *Divergent Change* smell, which refers to making unrelated changes in the same location. Fixing *Divergent Change* with means other than such established patterns might thus accidentally lead to *Feature Envy*. *Feature Envy* means many accesses to another class, e.g., in one method, which might lead to a *Long Method*. As often the same envious code is used in multiple places, *Feature Envy* can cause *Duplicate Code*.

4.3.3. Liabilities and principle violations

Practitioners often list either liabilities of a smell or principle violations to show why one should care about the smell (summarized in Table 4). Sometimes liabilities listed here are derived from positive aspects occurring when the smell is fixed. For *Feature Envy* a large fraction of the practitioners see *high coupling* (and its companion *low cohesion*) and *low code comprehensibility* as typical liabilities. Many practitioners see also various *maintainability issues* as liabilities such as *reduced changeability*, *missing encapsulation*, *misplaced responsibilities*, *reduced testability*, *reduced reusability*, and *high complexity caused by code duplication*.

Feature Envy can lead to violations of the *Single Responsibility Principle* when a method or class takes over responsibilities of another class (or its features) it is envious of. *Feature Envy* can lead to a violation of the *Tell Don't Ask Principle* which means that objects should not ask for details or internals, such as envied data features, but rather tell other objects what to do.

Note that the liabilities have dependencies, too, such as one liability can cause another one. We report the liabilities in this article in the way we found them in the sources, without considering such dependencies, as those dependencies are not discussed in our grey literature sources.

Smell	Liability/Principle Violation	Liability/Principle Violation Evidences
Feature Envy Number of evidences discussing liabilities/principles option: 26	High coupling	S1, S6, S7, S15, S18, S20, S26, S28, S30, S31, S43, S47
	Low cohesion	S6, S7, S26, S30, S43
	Low code comprehensibility	S4, S6, S11, S26, S29, S30, S31, S43, S47
	High complexity caused by code duplication	S4, S10, S43
	Reduced changeability	S6, S15, S20, S21, S26, S30, S31, S46
	Missing encapsulation	S10, S17, S18, S26, S27, S28, S29, S31
	Reduced testability	S10, S11, S28
	Reduced reusability	S11, S21, S26, S46
	Misplaced responsibility	S21, S27, S30, S31, S39, S43, S46
	Maintainability issues	S31, S38, S40
	Can cause violation of Single Responsibility Principle	S7, S15, S28, S29, S31
	Can cause violation of Tell Don't Ask Principle	S28
Inappropriate Intimacy Number of evidences discussing liabilities/principles option: 20	Relying on internal details can lead to defects	S1
	Maintainability issues	S2, S25, S31, S36, S38, S40
	High complexity	S2, S13
	Low code comprehensibility	S4, S13, S25, S26, S31, S36
	Hinders code reuse	S4
	Missing encapsulation	S2, S12, S13, S24, S26, S31
	Reduced changeability	S15, S24, S26
	High complexity caused by code duplication	S15
	High coupling	S15, S24, S25, S26, S31, S36
	Low cohesion	S24
	Reduced reusability	S26
	Can cause violation of Single Responsibility Principle	S15, S25, S26
	Can cause violation of Law of Demeter: Only talk to your immediate friends	S26, S44
Data Class Number of evidences discussing liabilities/principles option: 13	Low code comprehensibility	S4, S34
	High complexity caused by code duplication	S4
	High coupling	S8
	Low cohesion	S8
	Misplaced responsibility	S16, S32
	Reduced changeability	S34
	Can cause violation of Single Responsibility Principle	S7, S32, S34
	Can cause violation of Tell Don't Ask Principle	S7, S32
Indecent Exposure Number of evidences discussing liabilities/principles option: 11	High complexity	S5
	Low code comprehensibility	S5
	High coupling	S15
	Reduced changeability	S15
	Maintainability issues	S38
	Can cause violation of Single Responsibility Principle	S15
Message Chain Number of evidences discussing liabilities/principles option: 15	Low code comprehensibility	S4, S15
	High coupling	S4, S14, S15, S39
	Reduced changeability	S15, S36
	Maintainability issues	S40
	Can cause violation of Single Responsibility Principle	S15
	Can cause violation of Law of Demeter: Only talk to your immediate friends	S36, S39
Middle Man Number of evidences discussing liabilities/principles option: 13	Low code comprehensibility	S4, S15, S23
	High complexity	S15
	High coupling	S15, S23
	Reduced changeability	S15
	Missing encapsulation	S23
	Maintainability issues	S38, S40
	Can cause violation of Single Responsibility Principle	S15

Table 4: Liabilities/principle violations suggested per smell. Color coding indicates how many evidences of a smell containing liability/principle violations descriptions contain a specific liability/principle violations.

4.3.4. Related practices and patterns

As shown in Table 5, *Feature Envy* has numerous relations to various common practices and patterns. Many of them are additional reasons why (automated) detection and repair of *Feature Envy* can be very hard, as they describe situations that look like *Feature Envy* but are actually places in the code where *behavior is kept separate from its data for a clear purpose*. Examples of this are the *Visitor* [46] and *Strategy* [46] patterns, and practices of *self delegation*. Delegation classes or *Wrappers* such as in the *Adapter* [46], *Decorator* [46], or *Facade* [46] patterns are also places in the code where *Feature Envy* can be wrongly identified. Some sources mention *utility/helper functions or classes* as code that is similar to *Feature Envy* but is not *Feature Envy*. While none of these relations is mentioned by many sources, overall a lot of sources identify common coding practices and patterns that can be identical in appearance to *Feature Envy*. The relation of *Feature Envy* to inheritance is seen controversially; one source mentions that the *use of superclass features* might lead to a wrong identification of *Feature Envy*.

Quite a number of sources relate *Feature Envy* to Domain-Driven Design (DDD) [47]. Most of them mention the general *Domain Model* pattern [48] but some are mentioning more specific DDD patterns such as *Entity* [47], *Service* [47], *Value Object* [47], and *Ubiquitous Language* [47] that make up or represent the *Domain Model*. For DDD to have a positive effect on the code and throughout a software's evolution, it must be represented well in the code. *Feature Envy* can lead to the *Anemic Domain Model* anti-pattern[7], i.e., a *Domain Model* which does not combine data and process together in its realization. Thus fixing *Feature Envy* can greatly improve the realization of those pattern in the code.

In one source it is mentioned that a similar positive effect can also be achieved for the *Extension Methods* practice (i.e. methods added to existing types), and another one explains how the *Template Method* pattern [46] can be used to avoid *Feature Envy*.

4.3.5. Fix Options

As shown in Table 6 overall 28 sources discuss or list possible fixes for *Feature Envy*. Most of those are well-known refactorings. The most often suggested fix is the *move method* refactoring. In cases like the second example above before considering to move a method, the envious part of a method might have to be extracted with the *extract method* refactoring, which is also often suggested. Sometimes it makes sense to extract multiple parts at once and apply the *extract class* refactoring. E.g. when moving a method, *move field* might be needed as well.

It is a bit unclear from the rather generic definitions of the smell if *Feature Envy* can occur within an inheritance hierarchy. If this is considered, either *add higher-level or more abstract feature* or *subclass and extend* can be considered as fix options. Other options mentioned only by one source are *hide delegate*, *bundle multiple methods into a single method*, *hide internal details*, or to first *use OO metrics to detect smell* before applying other fix options.

The variety of fix options presented and the fact that *extract method* and other changes to a class might be needed before *move method* can be applied underline the points from above: While it can be easy for humans to spot *Feature Envy* in a method, automation of detection and fixing is hard. Just consider an example like the second *getTotalPrice()* example above, which is a *Long Method* with alternating and intertwined code fragments from the own class and multiple other classes the method is envious of. Refactoring in this situation requires multiple steps combining many of the fix options listed here in a creative way.

[7]See e.g. `https://martinfowler.com/bliki/AnemicDomainModel.html`.

Smell	Relation	Practice/Pattern	Smell-Practice Relation Evidences
Feature Envy	is not a smell if it is	Behavior is kept separate from its data for a clear purpose (sub practices/patterns: Visitor, Strategy, Self Delegation)	S4, S14, S16, S27
	is not a smell if it is	Visitor	S4, S14, S16, S27
	is not a smell if it is	Strategy	S4, S14, S16, S27
	is not a smell if it is	Self Delegation	S16
	might not be a smell if it is	Use of superclass features	S9
	is not a smell if it is	Wrapper	S9
	is not a smell if it is	Adapter	S42, S46
	is not a smell if it is	Decorator	S46
	is not a smell if it is	Facade	S42, S47
	might not be a smell if it is	Utility/Helper Function or Class	S6, S42
	can be avoided by use of	Template Method	S27
	fixing smell might improve	Domain Model	S10, S11, S17, S20, S21, S28
	fixing smell might improve	Entity	S10, S20
	fixing smell might improve	Service	S20
	fixing smell might improve	Extension Methods	S20
	fixing smell might improve	Value Object	S10, S11
	fixing smell might improve	Ubiquitous Language	S11
Inappropriate Intimacy	can be avoided by use of	Chain of Responsibility	S13
	might not be a smell if it is	Use of superclass features	S14
Data Class	is not a smell if it is	Data Transfer Object	S7, S8, S32, S34
	is not a smell if it is	Parameter Object	S7
	is not a smell if it is	Data Access Object	S32
	is not a smell if it is	Immutable Data Object	S16, S32
	fixing smell might improve	Domain Model	S17, S34
	fixing smell might improve	Service	S17
Middle Man	is not a smell if it is	Delegation class with clear purpose (sub practices/patterns: Adapter, Wrapper, Proxy, Facade, Decorator)	S3, S15, S22
	is not a smell if it is	Facade	S14, S15, S22
	is not a smell if it is	Wrapper	S3
	is not a smell if it is	Proxy	S15, S39
	is not a smell if it is	Adapter	S3, S15
	can evolve out of	Mediator	S14

Table 5: Practice and pattern relations suggested per smell. Color coding indicates how many evidences of a smell contain a specific practice or pattern.

Smell	Fix Option	Fix Option Evidences
Feature Envy Number of evidences discussing fixes option: 28	Move method	S1, S3, S4, S5, S9, S10, S11, S14, S15, S16, S17, S18, S20, S22, S27, S28, S29, S31, S35, S39, S42, S43, S45, S46, S47
	Add higher-level or more abstract feature	S1, S21, S27
	Extract method	S4, S5, S6, S9, S10, S11, S14, S16, S27, S28, S29, S31, S35, S39, S43, S45, S46, S47
	Extract class	S9, S14
	Move field	S5, S31, S35
	Hide delegate	S11
	Bundle multiple methods into a single method	S15
	Hide internal details	S20
	Use OO metrics to detect smell	S41
	Subclass and extend	S10, S39
Inappropriate Intimacy Number of evidences discussing fixes option: 14	Change target class to only offer public features if possible	S1
	Change calling class to only use public features if possible	S1
	Move method	S4, S5, S14, S22, S24, S25, S26, S35, S44, S48
	Move field	S4, S5, S14, S24, S25, S35, S44, S48
	Extract class	S4, S5, S24, S25, S44, S48
	Hide delegate	S4, S5, S24, S35, S48
	Replace delegation with inheritance	S4, S5, S24, S35, S44, S48
	Change Bidirectional Association to Unidirectional Association	S4, S5, S24, S25, S35, S48
	Use reflection with caution	S13
	Encapsulate field	S13, S15, S26
	Encapsulate collection	S13, S26
	Introduce extra class between intimate classes	S15
	Extract method	S22, S25, S26
	Subclass and extend	S10
Data Class Number of evidences discussing fixes option: 11	Move method	S4, S5, S7, S8, S14, S16, S17, S35
	Encapsulate field	S4, S5, S8, S14, S16, S35
	Encapsulate collection	S4, S5, S14, S16, S35
	Extract method	S4, S7, S14, S16, S17
	Move data-using methods to data class	S4, S8, S17, S32
	Remove getters and setters from data class and make fields private	S14, S16
	Make immutable data object	S32
	Use OO metrics to detect smell	S37
	Subclass and extend	S10
Indecent Exposure Number of evidences discussing fixes option: 1	Encapsulate field	S15
	Encapsulate collection	S15
	Hide behind method	S15
	Hide behind abstract class or interface	S15
Message Chain Number of evidences discussing fixes option: 9	Hide delegate	S4, S5, S14, S16, S22, S35, S39
	Extract method	S4, S5, S14, S16, S39
	Move method	S4, S5, S14, S16
	Ignore, if refactoring leads to middle man	S4
	Hide behind method	S15
	Ignore, if it is a small chain which causes little coupling (is harmless)	S14
	Subclass and extend	S10
Middle Man Number of evidences discussing fixes option: 12	Remove middle man	S4, S5, S14, S15, S16, S22, S23, S33, S35, S39
	Ignore, if middle man is used to reduce interclass dependencies	S4
	Inline method	S5, S16, S35
	Replace delegation with inheritance	S5, S35
	Remove middle man method	S19
	Subclass and extend	S10

Table 6: Fix options suggested per smell. Color coding indicates how many evidences of a smell containing fix option suggestions contain a specific fix option.

4.4. Inappropriate Intimacy

4.4.1. Definitions

Inappropriate Intimacy has the second highest number of evidences (21 sources) with five different definition variants in Table 3. Almost a half of the sources define *Inappropriate Intimacy* as a class using another class's implementation details, some interpret implementation details only as private features, and some others view only a single method as the coupling origin. Another type of definition sees *Inappropriate Intimacy* as two or more classes excessively depending on each other. Two of those sources again limit the definition to private features.

The following example briefly outlines how a class can make use of another class's implementation details. Here the method *getAccomodationType()* of the class *Tenant* uses the internal implementation detail of the class *Accomodation*. If *Accomodation* would make similar use of *Tenant's* features, the more narrow "two or more classes excessively depending on each other" kind of definition would be fulfilled, too.

```java
class Tenant {
  private String name;
  private Accomodation accomodation;
  public String tenantAccomodation = accomodation.type;
  // ..
  public String getAccomodationType(String newtype){
    accomodation.setType(newtype);
    if (tenantAccomodation.equals(accomodation.getType()))
      return tenantAccomodation;
    else
      return accomodation.getType();
   }
}
```

4.4.2. Relationships to other smells

As shown in Figure 4, *Object Orgy* is a synonym of *Inappropriate Intimacy* often used in the Perl community. We have confirmed this by studying a number of sources on *Object Orgy*. *Indecent Exposure* is very similar to *Inappropriate Intimacy* and fosters its occurrence. A couple of possible relations are mentioned only by one source: *Inappropriate Intimacy* can cause the *Duplicate Code* smell. *Inappropriate Intimacy* might be accompanied by *Feature Envy* as many intimate relations, as in our example above, might lead to classes using the other class excessively. Such intimate or excessive use can often be observed with *Data Classes*, which also can accompany *Inappropriate Intimacy*.

4.4.3. Liabilities and principle violations

As shown in Table 4, motivations for avoiding *Inappropriate Intimacy* are very often *maintainability issues* in general or more specific ones such as *low code comprehensibility*, *missing encapsulation*, *high coupling*, *high complexity* (maybe caused by code duplication), *reduced changeability*, *low cohesion*, and *reuse issues*.

Inappropriate Intimacy often violates the *Single Responsibility Principle* when responsibilities of another class are not delegated to the other class, but performed locally. It also violates the *Law of Demeter* ("only talk with your immediately friend") if classes make excessive use of many classes' implementation details and thus have more knowledge about other classes than necessary.

4.4.4. Related practices and patterns

As shown in Table 5, *Inappropriate Intimacy* has only a few typical links to practices and patterns. One source brings up the *Chain of Responsibility* pattern [46]. Its multiple handlers in a chain with clear interfaces can help to avoid coupling between the classes. Another source sees the use of superclass features as a harmless symptom only looking like *Inappropriate Intimacy*.

4.4.5. *Fix Options*

Inappropriate Intimacy has the widest range of fix options with 14 fix options from 14 sources as shown in Table 6. Strongly recommended fixes are *move method*, *move field*, *extract class*, *hide delegate*, *replace delegation with inheritance*, and *change bidirectional association to unidirectional association*. It is interesting to see that many of those also appear for *Feature Envy*. Again, the practitioner recommendation often describes fixing as a process in which many of those are applied to resolve a complex design situation. A couple of sources mention specific fixes for *Inappropriate Intimacy*, such as recommendation of increasing *encapsulation*, focus on *public features only*, *cautious use of reflection*, or *introducing an extra class between intimate classes*.

4.5. *Data Class*

4.5.1. *Definitions*

For *Data Class* all 16 sources covering the smell conform to the same kind of definition (as shown in Table 3): It is a class that only offers data. It might have methods but these are only getters and setters for the data. The following example shows such a pure *Data Class*:

```java
public class Student {
  private int id;
  private String name;

  public Student(int id){
    this.id = id;
  }
  public void setId(int id){
    this.id = id;
  }
  public void setName(int name){
    this.name = name
  }
  public int getId(){
    return id;
  }
  public String getName(){
    return name;
  }
}
```

4.5.2. *Relationships to other smells*

As shown in Figure 4, sometimes *Data Class* is accompanied by *Inappropriate Intimacy*, for instance, if other classes use a *Data Class's* private features or implementation details excessively. *Data Classes* can cause *Feature Envy* as classes using *Data Classes* might be envious to the data access features. If *Data Classes* are accessed via private or other internal features, they also suffer from *Indecent Exposure*, which can worsen the coupling of the *Data Class*. Using *Data Classes* entails the danger of causing *Duplicate Code*, if the functionality related to the data in the *Data Classes* is handled in several places across the code base.

4.5.3. *Liabilities and principle violations*

As shown in Table 4, *Data Classes* are harmful because they *lower the code comprehensibility*. They are seen as a *misplaced responsibility* if code manipulating the data is located elsewhere than the data. This is also the reason why *Data Classes* might violate the *Single Responsibility* and Tell Don't Ask Principles. Other liabilities often mentioned are other maintainability issues such as *high coupling*, *high complexity*, *low cohesion*, and *reduced changeability*.

4.5.4. Related practices and patterns

Data Class seems rather simple to detect because of its obvious symptoms. But this is deceiving. Practitioners point out many structurally and behaviorally more or less identical solutions, which are seen as best practices and not as harmful *Data Classes*. The most common of those is the *Data Transfer Object* [48] which is an object that carries data between processes, e.g. often used in distributed systems. Immutability is an important principle of functional programming, and thus *Immutable Data Objects* are used e.g. where objects and functional programming are combined. Complex parameters can be simplified with *Parameter Objects*. *Data Access Objects* are usually offering data access behavior as well, but some practitioners point out that they can be confused with *Data Classes*.

As explained above in Section 4.3.4, Domain-driven Design aims for rich *Domain Models* with data and behavior. *Data Class* can be seen as a symptom of the *Anemic Domain Model* anti-pattern, and fixing them can improve the *Domain Model* as well as the *Services* relying on domain model classes.

4.5.5. Fix Options

As shown in Table 6, reworking the various classes that use the *Data Class'* data by *move method*, *move data-using methods to data class*, and *extract method* are very often suggested fixes. In addition, information hiding fix option such as *encapsulate field*, *encapsulate collection* and *remove getters and setters from data class and make fields private* are often suggested, too. *Making data objects immutable* (i.e., turning then into *Immutable Data Objects*), *subclass and extend*, and *using OO metrics to detect the smell* before other fixes are applied are each suggested by one source.

4.6. Indecent Exposure

4.6.1. Definitions

For *Indecent Exposure* all 5 sources covering the smell agree about the definition (as shown in Table 3): It is a class that exposes internal detail. One typical option considered are classes offering public variables, e.g. in languages like Java. In other languages such as Python this might be seen as less problematic, but still internal details can be exposed. Finally, many other language options exist, e.g. reflection or pointer manipulations, that might lead to *Indecent Exposure*.

4.6.2. Relationships to other smells

As shown in Figure 4, *Excessive Exposure* is another name of *Indecent Exposure*. *Indecent Exposure* is similar to *Inappropriate Intimacy* and can enable its occurrence. *Indecent Exposure* makes *Data Classes* worse because it reveals their data without any restrictions.

4.6.3. Liabilities and principle violations

For *Indecent Exposure maintainability issues* such as *high complexity*, *low code comprehensibility*, *high coupling*, and *reduced changeability* are discussed as typical liabilities. *Indecent Exposure* might lead to violations of the *Single Responsibility Principle*, if other classes use the exposed features to realize responsibilities belonging to those features.

4.6.4. Fix Options

Table 6 shows 4 different fix options for *Indecent Exposure*. They are all about adding more encapsulation or information hiding: *encapsulate field*, *encapsulate collection*, *hide behind method*, and *hide behind abstract class or interface*.

4.7. Message Chain

4.7.1. Definitions

For *Message Chain* our 12 sources agree on a single definition: long sequence of method calls. They often distinguish a call chain, as in the following example, or reaching them same with other means, e.g. by using temporary variables.

```
int id = obj.getDepartment().getSubDepartment().getHOD().getId();
```

4.7.2. Relationships to other smells

Fixing *Message Chain* can cause the *Middle Man* smell, if an object is introduced to only coordinate the calls in the chain.

4.7.3. Liabilities and principle violations

For *Message Chain* various *maintainability issues* such as *high coupling*, *low code comprehensibility*, and *reduced changeability* are discussed as typical liabilities. In a *Message Chain* calls might combine various responsibilities in one place, violating the *Single Responsibility Principle*. *Message Chains* might lead to more linked objects through calls, i.e. a violation of the *Law of Demeter*.

4.7.4. Fix Options

The majority of fix options are about step-wise reworking the *Message Chain*, i.e. *hide delegate*, *extract method*, *move method*, *hide behind method*, and *subclass and extend*. It is advised to ignore the smell, if it leads to introduction of a *Middle Man* or if it is just a *small chain causing little coupling*.

4.8. Middle Man

4.8.1. Definitions

Middle Man appears in 13 sources shown in Table 3. More than a half of the sources define *Middle Man* as a class that only delegates to other classes. Similarly, another definition sees the *Middle Man* as a Wrapper. Some practitioners mention that a *Middle Man* might do other work than only delegating. This is reflected in one definition variant by stating that the majority of the work is delegated. In another one it is introduced by making a single method the origin of the coupling, i.e. this way other methods of the class can perform other tasks than pure delegation.

4.8.2. Relationships to other smells

As shown in Table 2, *Middle Man* can result after resolving *Message Chain*.

4.8.3. Liabilities and principle violations

For *Middle Man* various *maintainability issues* such as *low code comprehensibility*, *high complexity*, *high coupling*, *reduced changeability*, and *missing encapsulations*, are discussed. A *Middle Man* gathers responsibilities that likely belong to the wrapped classes, i.e. it often violates the *Single Responsibility Principle*.

4.8.4. Related practices and patterns

There are structurally and behaviorally identical situations to *Middle Man* which are not considered as a smell. They occur if it is a *delegation class with a clear purpose*. Examples might be patterns and practices such as *Facade* [46], *Wrapper*, *Proxy* [46], and *Adapter* [46]. A *Mediator* [46] is also structurally and behaviorally similar, and if its mediation responsibilities shrink during evolution, it might turn into a mere *Middle Man*.

4.8.5. Fix Options

Most practitioners agree to resolve the *Middle Man* smell by a simple *remove middle man* fix (*remove middle man method* is similar to this). Three sources suggest to fix it by *inline method*. Two sources suggest to fix it by reworking the calls into the inheritance hierarchy: *replace delegation with inheritance* and *subclass and extend*. One source suggests *ignore if middle man is used to reduce interclass dependency*.

5. Discussion

5.1. Discussion of how practitioners understand coupling smells

In the previous section, we have detailed most of our finding regarding **RQ1**, i.e., our derived model how coupling smells are understood by practitioners. In particular, by describing our meta-model, depicted in Figure 2, we have outlined some of our findings about **RQ1.1**, i.e. what the relevant defining factors of coupling smells are. This is detailed in Figure 3 with possible relationship types. In addition, we have categorized the definitions of the smells used by practitioners for each of the coupling smells, as well as the coupling origins and targets in those definitions (see Table 3). Here, it is interesting to observe that many practitioners have a rather broad and hard to formalize understanding of the smells (such as "excessive use" based definitions). This means, it requires design expertise to judge whether or not a certain situations is a smell, making automated smell detection hard. While most coupling smells have only small variations in their definitions, the scope of *Feature Envy* is more controversial.

It is worth to note that the practitioners sometimes confuse the various coupling smells. For instances, a confusion between *Feature Envy* and *Inappropriate Intimacy* is mentioned in S1. Some *Feature Envy* definitions actually describe *Data Envy* (e.g. in S28). *Feature Envy* can also be confused with more simple structures, e.g. utility functions (see e.g. S6, S42). Some practitioners see a difference between the notions of *Indecent Exposure* and *Inappropriate Intimacy*, for others it is the same smell.

RQ1.2 considers what the relevant quality impacts and trade-offs are. We have mainly found the stated liabilities and principle violations (see Table 4). Across the different smells, *low code comprehensibility*, i.e. impact on understandability qualities, is often mentioned, as well as various kinds of *maintainability issues* affecting software qualities such as understandability, changeability, evolvability, testability, and reusability. Violations to *Single Responsibility*, *Tell Don't Ask*, and *Law of Demeter* principles are often reported, too (and closely related to the mentioned liabilities). In a very few cases, impact on *Code* and *Design Technical Debt* are discussed, as well. We have found no evidence for concrete ways to spot, calculate, or pay back technical debt other than smell definitions, fixes, and two sources mentioning simple OO metrics for detection (S37, S41).

RQ1.3 investigates what the relevant relations among smells and to other software concepts are. We have found relations between smells (see Table 2), as well as relations to patterns and practices (see Table 5). For many coupling smells structurally and behaviorally identical solutions exist, which are considered in the literature as best practices. For example, for *Data Class* various exceptions to the definition are discussed, which are actually best practices, that some practitioners doubt whether *Data Class* is actually a code smell (see e.g. S32). For this reason, *Data Class* is, even though structurally very simple, rather hard to automatically detect without considering human design expertise.

A remarkable observation is that many practioners relate Domain-Driven-Design to coupling smells. It seems a good *Domain Model* design can help to avoid coupling smells, and existing coupling smells might also need to be considered at the *Domain Model* level to be fixed well.

An overarching aspect concerning **RQ1.2** and **RQ1.3** is that simple examples of the smells, such as those used for illustration in this article, might be not critical enough to be considered harmful. Rather with enough *intensity of the smell*, a harmful negative effect on the qualities can arise that requires fixing.

For instance, if multiple smells occur together, the intensity of each single smell might increase. For that reason, understanding the relations of smells is important.

RQ1.4 studies what the relevant options for fixing coupling smells are. We have found the detailed lists of fixes outlined in the previous section and summarized in Table 6. Here, it is interesting to observe that for situations where the smells occur in enough intensity to be harmful, the situations will likely also be too complex to be fixed by a single fix and often there are many options available for fixing. Complex interdependencies to other places in the code need to be considered for fixing the smells by applying a cascade of refactorings or other fix options.

Overall, the relation of many smells to inheritance practices remains unclear to a certain extent. Also, if inheritance can or should be used to fix the smells remains unclear. Smells might differ in different programming languages. For example, practices considered as *Indecent Exposure* in Java might be acceptable to Python developers. The developer's perception is also dependent on organization standard and project size. For example, S40 states that: "Code smell differs from project to project and developer to developer, according to the design standards that have been set by an organization." Our results indicate that understanding the smells' contexts, such as programming languages, project details, and design methods, could help understanding the implications of smells better. More research is needed to understand the impact of the smells' contexts.

For all of RQ1, it is interesting that practitioners sources often relate to only a few core sources, especially Fowler's refactoring book [1],

5.2. Interesting gaps to the scientific literature and future research opportunities and challenges

In this section we discuss the results in the relation to **RQ2**, i.e., what the gaps in the understanding of coupling smells between the practitioner's view and the scientific literature are. From each discussed aspect we derive lessons as answers to **RQ3**, i.e., what the resulting opportunities and challenges for future scientific work are.

5.2.1. Coupling smell definitions and their detection

At first glance, the definitions of coupling smells used in research literature and those in the practitioner texts seem to match well. At the closer inspection performed in our study, this is not the case. Let us illustrate this using the example of *Feature Envy*: The scientific detection literature summarized in Section 2 mostly uses a definition such as "a method (class) uses more features (data) of another class than its own" in very strict sense, as this is easily measurable e.g. with distance metrics. As outlined, many practitioners rather use broader, pragmatic definitions. Practitioners seem to assume some code can be classified as *Feature Envy* only if a substantial harmful impact on desired qualities is present. The relations to practices and patterns found in our study reveal many exceptions that must be made to the definitions of coupling smells. Overall, researchers rarely take patterns and practices into account [2] and if so then only by other simple heuristics. For example, there is no structural or behavioral clue if some code detected as having a *Feature Envy* like structure and/or behavior is not really a *Visitor* or *Strategy* pattern. In response, some approaches tailor their detection heuristics by looking for terms like "visitor" and "strategy" in class and package names, but this approach is futile: Firstly, there is no guarantee that a visitor instance has "visitor" in its name. Secondly, these patterns are only used as examples by practitioners for situations where *behavior is kept separate from its data for a clear purpose*. There are many other such cases that look like *Feature Envy*, but are neither *Feature Envy* nor *Visitor* or *Strategy*. The found relations to domain-driven design practices reveal many additional situations where *Feature Envy* can be harmful; thus for detecting *Feature Envy* well, often a deep understanding of the system's *domain model* is required, too. Finally, often *Feature Envy* is seen as being harmful only if it occurs intensely enough in relation with other smells. The empirical study by Palomba et al. [17] confirms this finding. In other words, an approach to detect *Feature Envy* could consider all rich relations to the other smells that *Feature Envy* has (see Figure 4) into account.

While *Feature Envy* is the most complex smell in our study, the situation is more or less the same for all other coupling smells. As a consequence, many of the smell detection results found by current scientific tools would be considered not being the smell or non-harmful occurrence of the smell by practitioners, for one or the other of the reasons given above. Note that a couple of the empirical studies with human participants in Section 2.3 report similar findings (e.g. [16, 15, 18, 19]). For example, Carneiro et al.'s study [15] confirms our result that practitioners see coupling smell detection as a complex problem, as opposed to the tools which only use simple heuristics. Mantyla et al.'s study [16] confirms this view, too, e.g. as they found that conflicting perceptions of different evaluators play a large role. In Mantyla et al.'s study, metrics-based and practitioner identification of smells do not correlate, a result which our study would predict as a possible result. Guo et al. [18] found that semantic factors of smells could not be encoded into their tool, which can also be explained using our results. Yamashita and Moonen [19] found that interaction effects amongst collocated smells and coupled smells should be taken into account. This is confirmed by our study, revealing many inter-smell relations. This study has also found that smells alone cannot predict maintainability issues. This is not surprising according to our results, which indicate that a considerable number of other aspects need to be taken into account when judging smells.

Lesson 1. *The understanding of coupling smells used in scientific studies could benefit from being broadened: When classifying some code as a coupling smell, it would make sense to consider a qualitative assessment of the code's impact on code and design qualities, the relations and interactions with other smells, and the relations to practices, patterns, and the system's domain model that sometimes lead to exceptions or changed quality impacts.*

As outlined in Section 2.1, many of the current coupling smell detection approaches make the assumption that coupling smells can be detected by rather simple heuristics such as metrics, static or dynamic analyses, or metrics-based rules. Broader understandings of coupling smells, as suggested above, mean that no heuristic technique can ever take all situations covered by those definitions into account. That is, heuristics can help to find suspicious places in the code, but cannot solidly make the decision if it is a code smell or not.

Lesson 2. *Scientific studies could stronger focus on providing hints to developers and getting human design and domain expertise into the loop before classifying some code as being a coupling smell.*

Given the typical work pressures of industrial software developers, we speculate that using localized approaches to provide such hints for smell candidates directly during their work on the code, e.g. in the IDE or code editor, is advisable, instead of tools providing long lists of potential code issues (which contain a lot of false positives).

5.2.2. Coupling smell fix options

A similar situation as for the detection can actually be observed for the fixes of coupling smells. As can be seen (see e.g. Table 6), practitioners consider a much broader variety of fix options than usually considered in the scientific literature discussed in Section 2. Also they often consider doing many more changes than just one single refactoring. For example, whereas the scientific literature sometimes seems to imply typical harmful *Feature Envy* can be easily fixed by moving a method, very often a much more complex procedure is needed. Consider a complex, central class of a system envious to many other classes, but intertwined in many ways with those other classes. According to the various practitioner sources many steps for fixing might be needed. Our GLS results indicate that fixing a complex coupling situation can be a complex multi-criteria optimization or decision problem that needs deep domain and design experience to be solved. It is thus explainable that empirical studies where human participants review small code entities lead to the result that practitioners do not see the smell identified by the researchers as being problematic (as e.g. in [17]).

Lesson 3. *Scientific studies could take into account a broad range of fixing options for a coupling smell and the trade-offs between various fixing options. They could consider the impacts on other smells, practices, patterns, the system's domain model, relevant code and design qualities, and so on. Approaches could consider getting human design and domain expertise into the loop. They could see fixing a highly coupled design situation as a complex multi-criteria optimization or decision problem.*

5.2.3. Domain-specific and other specializations for coupling smells and related approaches

It is interesting to observe that practitioners explicitly or implicitly discuss the relation to domain modeling, domain-driven design, and related semantically rich areas of software design quite often. In the scientific literature discussed in Section 2, these aspects play a very minor role. To explore such relations could lead to a better understanding of the rich semantic understanding of coupling smells revealed in our GLS .

Lesson 4. *Scientific studies could start to study the relations of domain modelling, domain-driven design, and related areas to coupling smells more intensely.*

Domain modelling and domain-driven design are areas, where more knowledge about the system or domain can help to improve smell detection and fixing. We have summarized a number of scientific approaches already going into that direction in Section 2.2. This is an interesting research direction that could be intensified. Many aspects identified in our study as being rather broad in the view of the practitioners, such as impact of liabilities and principle violations, relations to practices and patterns, inter-smell relations, and fix options, might be just that broad because the coupling smells cover every possible domain and every kind of system. Limiting the view e.g. to software architecture, microservice-based systems, domain modelling aspects, or enterprise architectures might greatly reduce the breadth of options and at the same time lead to more specific smells. Guo et al.'s empirical study results [18] confirm this and suggest to tailor the generic heuristics usually used for detection to take domain-specific factors into account.

Lesson 5. *Scientific studies could consider more specialized coupling smells and related approaches, for instance, in certain application or technology domains, or consider the smells only in a specific (technical) context.*

Note that covering more application or technology domains, and other such specializations, requires broader views on coupling smells (as suggested by Lesson 1). Very narrow understandings, e.g. in automated tools, might miss application- or technology-specific smells and fix options.

5.3. Experiences, Challenges, and Lessons Learned in the Grey Literature Study

A secondary contribution of our work is the integration of GT with GLS, using the grey literature as a data source for GT. In our experience, this combination works well for studying existing phenomena in practice and their relations. For instance, it is very well applicable for studying which smells practitioners are concerned with, and their relations to existing patterns and practices, quality attributes (liabilities), principles, inter-smell relations, and so on.

A benefit of using the grey literature compared to interviews, which are often used as a data source in GT, are reduced biases: avoiding the interview situation context and leading the interviewees in the expected direction of interviewers. However, a downside of this is that practitioner sources are not written with the goal to fuel a GLS. Thus, a major challenge is that identifying gaps between scientific and practitioner understandings is always prone to a certain level of interpretation by the researchers. To address this challenge we opted to just report opportunities and challenges for research, rather than criticizing existing research directions. Stronger statements would require additional research, e.g. quantitative studies or surveys, showing that the observed phenomena are relevant for practitioners at a broader scale. Multi-vocal literature reviews might also be a better option for research in such directions. All those research methods, however, would lack the explorative nature of our study.

In other GLSs and especially in multi-vocal literature reviews, the search strategy for the grey literature plays a central role. This is minimized in GT-based research, as GT does not aim to study phenomena holistically, but rather phenomena that have specifically been observed to exist [49]. This however means that reporting the literature search strategy in an overview is harder than for other GLS. To fully understand the search strategy, each coding step must be studied. To enable this, we provide all data and code in an open access repository. This downside of GT has a big advantage nonetheless: In most other GLSs it is hard to define exclusion criteria to avoid having to include sources of low quality. For example, in our study we could exclude otherwise well written sources that aimed to sell a product or service by a qualitative review performed by the authors. That is, the selection process is very similar to interview studies and other such qualitative research – and has the same threats to validity discussed below.

We combined the GLS with UML-based, rigorous modelling for coding because it is highly beneficial to precisely represent the results in our experience. A major challenge is that the grey literature usually consists of rather informal writings and thus any categorization or classification performed by the researchers has a subjective element to it. Yet, this is the case for all axial coding in GT; it is not a specific property of our study. In constract to text-based axial coding, the Python implementation keeps track of relations between codes, auto-layouts visualizations of those relations, and yields an error for many coding mistakes researchers might make. Such errors might go undetected in text-based coding.

Another big difference to many systematic literature studies is that each included source was not only scanned for knowledge, but instead analysed in-depth, line-by-line, usually many times. Whenever a later studied source reveals new insights on a phenomenon studied before, we re-analyzed the related source studied before. Here, our experience shows that precisely coded trace links can help a lot in guiding which sources provided evidence for some phenomenon and thus might need to be inspected again. To enable replicability of such studies, it is important to fully document each coding stage for each source – as provided in our open access replication package.

6. Threats to validity

It is important to consider the threats to validity during the design of a study to increase its validity. As GT is mainly concerned with phenomena that have specifically been observed to exist, threats to the results' validity are mainly restricted to inappropriate conceptualization [49]. As we carefully added practitioner articles until theoretical saturation was reached, and then carefully reviewed and coded each source in multiple iterations by all authors, we believe the validity of our results as likely to be high. We do not claim that all practitioner in our data sources are experts for coupling smells. Our study has a certain mono-method bias as it uses grounded theory for analyzing grey literature only. To mitigate the threat we contrasted our finding in-depth to the scientific literature.

To increase internal validity or credibility we decided to use practitioner reports that were produced independent of our study. This avoids bias, e.g. compared to interviews in which the practitioners would have known that their answers are used in a study. However, this introduces a different threat: Some important information might be missing in the reports, which would have been revealed in an interview. We tried to mitigate this threat by looking at many more sources than needed to reach theoretical saturation, as it is unlikely that all different sources miss the same important information.

The different members of the author team have cross-checked all models independently to minimize researcher bias. The threat to validity that the researcher team is biased in some sense remains, however. The same applies to our coding procedure and the UML-based modeling: Other researchers might have coded or modelled differently, leading to different models. As our goal was only to find one model that is able to specify all observed phenomena, and this was achieved, we consider this threat not to be a major issue for our study.

The experience and search-based procedure for finding knowledge sources may have introduced some kind of bias as well. However, this threat is mitigated to a large extent by the chosen research method, which requires just additional sources corresponding to the inclusion and exclusion criteria, not a specific distribution of sources. Note that our procedure is in this regard rather similar to how interview partners are typically found in qualitative research studies in software engineering today. However, the threat remains that our procedures introduced some kind of unconscious exclusion of certain sources; we mitigated this by assembling an author team with many years of experience in the field, and performing very general and broad searches.

GT aims to explain phenomena that exist, and it neither claims to capture all such phenomena nor to quantify their frequency or distribution [49]. Therefore, even results derived from a very few sources will be valid and can be relevant [49]. There is a risk that generalizing from some of our results might be misleading, but as we do not claim completeness and use a rather high number of sources (many more than needed for theoretical saturation), it is likely that generalization is valid to at least some extent. Note that we have provided the number of sources mentioning a phenomenon to give an impression of the data we gathered; this should not be misinterpreted as quantification of frequency or distribution.

7. Conclusions

Based on a Grounded Theory study of grey literature sources containing practitioner views on coupling smells, this article has found defining factors of coupling smells and relevant impacts on and trade-offs to be made with regard to code and design quality. We identified Feature Envy, Inappropriate Intimacy, Data Class, Indecent Exposure, Message Chain, and Middle Man as coupling smells widely discussed by practitioners. We further studied the relations among those coupling smells, to other related smells, and to other software code and design concepts, as well as options for fixing coupling smells. Based on this knowledge, we identified gaps in the understanding of coupling smells between science and practice. Finally, we derived opportunities and challenges for future scientific work. In particular, we have derived five lessons that represent opportunities and challenges for future research. From those we can conclude that the definitions of coupling smells used in approaches and tools need to be broadened to the various practitioner concerns identified in this article. Coupling smells detection tools and approaches could aim to better include human design expertise in the detection and fixing decisions on coupling smells. Coupling smell fixing often could be understood as a complex multi-criteria optimization or decision problem, not a simple application of one or a few refactorings. The interaction of coupling smells and domain modelling could be studied more intensely. As future work, we plan to examine more closely some of the lessons learned in our study, especially more specialized and domain-driven approaches. We also plan to confirm some of the lessons learned in other empirical studies. It would be interesting to perform further quantitative studies specifically focused on the relations of smells to practices, liabilities, fixes, and so on, which we have found in this article.

[1] Refactoring: Improving the Design of Existing Code, Addison-Wesley Longman Publishing Co., Inc., USA, 1999.

[2] F. Sabir, F. Palma, G. Rasool, Y.-G. Guéhéneuc, N. Moha, A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems, Software: Practice and Experience 49 (1) (2019) 3–39.

[3] M. Lanza, R. Marinescu, Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems, Springer, 2006.

[4] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, Jdeodorant: Identification and removal of feature envy bad smells., in: ICSM, IEEE Computer Society, 2007, pp. 519–520.

[5] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, A. De Lucia, Mining version histories for detecting code smells, IEEE Transactions on Software Engineering 41 (5) (2015) 462–489.

[6] A. M. Fard, A. Mesbah, Jsnose: Detecting javascript code smells, in: 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2013, pp. 116–125.

[7] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, A. Marino, Comparing and experimenting machine learning techniques for code smell detection, Empirical Softw. Engg. 21 (3) (2016) 1143–1191.

[8] H. Liu, Z. Xu, Y. Zou, Deep learning based feature envy detection, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 385–396. doi:10.1145/3238147.3238166.

[9] N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities, IEEE Trans. Softw. Eng. 35 (3) (2009) 347–367. doi:10.1109/TSE.2009.1.

[10] V. Sales, R. Terra, L. F. Miranda, M. T. Valente, Recommending move method refactorings using dependency sets, in: 2013 20th Working Conference on Reverse Engineering (WCRE), IEEE, 2013, pp. 232–241.

[11] R. Terra, M. T. Valente, S. Miranda, V. Sales, Jmove: A novel heuristic and tool to detect move method refactoring opportunities, Journal of Systems and Software 138 (2018) 19 – 36. doi:https://doi.org/10.1016/j.jss.2017.11.073.

[12] A. K. Dipongkor, I. Ahmed, N. Nahar, Move method recommendation using call frequency of methods and attributes, in: 2018 Joint 7th International Conference on Informatics, Electronics & Vision (ICIEV) and 2018 2nd International Conference on Imaging, Vision & Pattern Recognition (icIVPR), IEEE, 2018, pp. 76–81.

[13] M. Mantyla, J. Vanhanen, C. Lassenius, A taxonomy and an initial empirical study of bad smells in code, in: International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings., 2003, pp. 381–384.

[14] R. Marticorena, C. López, Y. Crespo, Extending a taxonomy of bad code smells with metrics, in: Proceedings of 7th International Workshop on Object-Oriented Reengineering (WOOR), Citeseer, 2006, p. 6.

[15] G. d. F. Carneiro, M. Silva, L. Mara, E. Figueiredo, C. Sant'Anna, A. Garcia, M. Mendonça, Identifying code smells with multiple concern views, in: 2010 Brazilian Symposium on Software Engineering, 2010, pp. 128–137.

[16] M. V. Mantyla, J. Vanhanen, C. Lassenius, Bad smells - humans as code critics, in: 20th IEEE International Conference on Software Maintenance, 2004. Proceedings., 2004, pp. 399–408.

[17] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, Do they really smell bad? a study on developers' perception of bad code smells, in: 2014 IEEE International Conference on Software Maintenance and Evolution, 2014, pp. 101–110.

[18] Y. Guo, C. Seaman, N. Zazworka, F. Shull, Domain-specific tailoring of code smells: An empirical study, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE '10, Association for Computing Machinery, New York, NY, USA, 2010, p. 167–170. doi:10.1145/1810295.1810321.

[19] A. Yamashita, L. Moonen, To what extent can maintenance problems be predicted by code smell detection?–an empirical study, Information and Software Technology 55 (12) (2013) 2223–2242.

[20] B. G. Glaser, A. L. Strauss, The Discovery of Grounded Theory: Strategies for Qualitative Research, de Gruyter, New York, NY, 1967.

[21] J. Corbin, A. L. Strauss, Grounded theory research: Procedures, canons, and evaluative criteria, Qualitative Sociology 13 (1990) 3–20.

[22] A. Rainer, A. Williams, Using blog-like documents to investigate software practice: benefits, challenges and research directions, Journal of Software: Evolution and Process (8 2019). doi:10.1002/smr.2197.

[23] U. Zdun, M. Stocker, O. Zimmermann, C. Pautasso, D. Lübke, Guiding architectural decision making on quality aspects in microservice apis, in: C. Pahl, M. Vukovic, J. Yin, Q. Yu (Eds.), Service-Oriented Computing - 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings, Vol. 11236 of Lecture Notes in Computer Science, Springer, 2018, pp. 73–89. doi:10.1007/978-3-030-03596-9_5.

[24] U. Zdun, E. Ntentos, K. Plakidas, A. E. Malki, D. Schall, F. Li, On the design and architecture of deployment pipelines in cloud- and service-based computing - A model-based qualitative study, in: E. Bertino, C. K. Chang, P. Chen, E. Damiani, M. Goul, K. Oyama (Eds.), 2019 IEEE International Conference on Services Computing, SCC 2019, Milan, Italy, July 8-13, 2019, IEEE, 2019, pp. 141–145. doi:10.1109/SCC.2019.00033.

[25] A. E. Malki, U. Zdun, Guiding architectural decision making on service mesh based microservice architectures, in: T. Bures, L. Duchien, P. Inverardi (Eds.), Software Architecture - 13th European Conference, ECSA 2019, Paris, France, September 9-13, 2019, Proceedings, Vol. 11681 of Lecture Notes in Computer Science, Springer, 2019, pp. 3–19. doi:10.1007/978-3-030-29983-5_1.

[26] E. Ntentos, U. Zdun, K. Plakidas, D. Schall, F. Li, S. Meixner, Supporting architectural decision making on data management in microservice architectures, in: T. Bures, L. Duchien, P. Inverardi (Eds.), Software Architecture - 13th European Conference, ECSA 2019, Paris, France, September 9-13, 2019, Proceedings, Vol. 11681 of Lecture Notes in Computer Science, Springer, 2019, pp. 20–36. doi:10.1007/978-3-030-29983-5_2.

[27] A. Chatzigeorgiou, A. Manakos, Investigating the evolution of bad smells in object-oriented code, in: 2010 Seventh International Conference on the Quality of Information and Communications Technology, 2010, pp. 106–115.

[28] N. Tsantalis, T. Chaikalis, A. Chatzigeorgiou, Ten years of jdeodorant: Lessons learned from the hunt for smells, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018, pp. 4–14.

[29] F. Palomba, D. Di Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, A. De Lucia, Landfill: An open dataset of code smells with public evaluation, in: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, 2015, pp. 482–485.

[30] M. M. Rahman, R. R. Riyadh, M. R. Rahman, Recommendation of move method refactorings using coupling, cohesion and contextual similarity, in: 2017 IEEE International Conference on Imaging, Vision & Pattern Recognition (icIVPR), IEEE, 2017, pp. 1–6.

[31] F. A. Fontana, V. Ferme, M. Zanoni, Towards assessing software architecture quality by exploiting code smell relations, in: 2015 IEEE/ACM 2nd International Workshop on Software Architecture and Metrics, 2015, pp. 1–7.

[32] S. Vidal, E. Guimaraes, W. Oizumi, A. Garcia, A. D. Pace, C. Marcos, Identifying architectural problems through prioritization of code smells, in: 2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS), 2016, pp. 41–50.

[33] J. Garcia, D. Popescu, G. Edwards, N. Medvidovic, Identifying architectural bad smells, in: Proc. of the 13th European Conference on Software Maintenance and Reengineering (CSMR), 2009, pp. 255–258.

[34] D. Taibi, V. Lenarduzzi, On the definition of microservice bad smells, IEEE software 35 (3) (2018) 56–62.

[35] D. Ratiu, S. Ducasse, T. Gîrba, R. Marinescu, Using history information to improve design flaws detection, in: Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04), CSMR '04, IEEE Computer Society, USA, 2004, p. 223.

[36] C. Marinescu, Identification of design roles for the assessment of design quality in enterprise applications, in: 14th IEEE International Conference on Program Comprehension (ICPC'06), 2006, pp. 169–180.

[37] F. Arcelli Fontana, V. Ferme, M. Zanoni, Filtering code smells detection results, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 2, 2015, pp. 803–804.

[38] R. B. Johnson, L. Christensen, Educational research: Quantitative, qualitative, and mixed approaches, SAGE Publications, Incorporated, 2019.

[39] M. Kenny, R. Fourie, Contrasting classic, straussian, and constructivist grounded theory: Methodological and philosophical conflicts, Qualitative Report 20 (2015) 1270–1289.

[40] V. Garousi, M. Felderer, M. V. Mäntylä, A. Rainer, Benefitting from the grey literature in software engineering research, CoRR (2019).
URL `http://arxiv.org/abs/1911.12038`

[41] V. Garousi, M. Felderer, M. V. Mäntylä, Guidelines for including grey literature and conducting multivocal literature reviews in software engineering, Inf. Softw. Technol. 106 (2019) 101–121. doi:10.1016/j.infsof.2018.09.006.

[42] C. Hagstrom, S. Kendall, H. Cunningham, Googling for grey: Using google and duckduckgo to find grey literature, in: 23rd Cochrane Colloquium. Cochrane database systematic reviews supplements, 2015, pp. 1–327.

[43] J. Piasecki, M. Waligora, V. Dranseika, Google search as an additional source in systematic reviews, Science and engineering ethics 24 (2) (2018) 809–810.

[44] M. V. Mäntylä, C. Lassenius, Subjective evaluation of software evolvability using code smells: An empirical study, Empirical Software Engineering 11 (3) (2006) 395–431.

[45] K.-J. Stol, P. Ralph, B. Fitzgerald, Grounded theory in software engineering research: a critical review and guidelines, in: Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 120–131.

[46] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[47] E. Evans, Domain-Driven Design: Tacking Complexity In the Heart of Software, Addison-Wesley Longman Publishing Co., Inc., USA, 2003.

[48] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley Longman Publishing Co., Inc., USA, 2002.

[49] F. Zieris, L. Prechelt, On knowledge transfer skill in pair programming, in: Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM'14, Association for Computing Machinery, New York, NY, USA, 2014. doi:10.1145/2652524.2652529.