Recovering Domain Knowledge using Formal Concept Analysis

Christian Köppe¹, Michael W. Godfrey², Jurgen J. Vinju³
¹Hogeschool Utrecht, Netherlands
²,³Centrum Wiskunde & Informatica, Amsterdam, Netherlands
¹christian.koppe@hu.nl, ²migod@cwi.nl, ³jurgen.vinju@cwi.nl

Abstract—We want to facilitate reengineering of families of (legacy) object-oriented applications to software written using domain specific languages. The hard problem is design of the domain specific languages. Good language design does not scale easily, so we would like to reuse existing knowledge.

What information from a legacy application can be the starting point of a design for a domain specific language? We focus on source code as a primary source of domain knowledge. The first questions are: "What domain concepts are relevant?" and "How can we filter the domain concept identifiers from the source code?"

We experimented with formal concept analysis to characterize domain concept identifiers. We study the shape of concept lattices to learn where and how domain concept identifiers appear. This leads to a proposed algorithm for domain concept recovery. Initial experiments using a prototype tool, called DoKRe, indicate promising results.

Keywords—TODO

1. INTRODUCTION

We are interested in re-engineering legacy object-oriented systems to systems that employ domain specific languages (DSLs). "Object-oriented systems are the legacy systems of tomorrow" [7]. The promise is that DSLs provide a more flexible and high-level way of dealing with variation within and evolution of families of software artifacts. Designing a domain specific language is hard though. First and foremost, the designers require an accurate and complete understanding of the respective domains.

In general accurate domain knowledge is a crucial factor when re-engineering legacy systems. Most probably, all that remains constant between the old design and the new design is the domain the systems reside in. While requirements change, architecture changes, implementation technology changes, the domain should remain relatively constant. Moreover, most of the value of a system is not in how it is implemented, but what it actually implements. The question is, given lacking documentation, decades of incremental development, and original designers having moved on, where to find this domain knowledge? Domain knowledge recovery ([7], [7]) is the discipline of making knowledge which is implicitly contained in existing software, explicit in the form of a domain description.

At least hypothetically, domain knowledge should be embedded in the source code of the legacy systems. Practically though, it seems a daunting task to recover such knowledge. It is almost paradoxical: The older a system is the more knowledge it must represent, but all the more hard it will be to recover this knowledge [7], [7], [7]. In this paper we try nevertheless, and focus on recovering knowledge from source code while ignoring other sources of information. The source code has one big advantage over other sources of information: it truly represents the factual state of the systems under investigation.

A. Class diagrams as domain descriptions

A general and precise definition of domain knowledge is not in the scope of our field of expertise. We assume that domain knowledge is everything a software designer needs to know about a domain before designing (families of) software systems. Instead we focus on domain descriptions. A domain description is a set of terms that denote important concepts of a domain and a set of relations between these terms form the basis of a domain description. As such a UML class diagram could be used to denote this form of domain knowledge.

A good domain description is: (a) complete, such that all relevant terms and relations are mentioned, (b) consistent, such that no two terms for the same concept are used and no two different relations between terms are used for the same conceptual relation, and (c) concise, such that no irrelevant terms and irrelevant relations are included.

Given the size and complexity of the source code of legacy software systems, it is a challenge to try and find out whether we could fully automatically extract and verify such domain descriptions. This is our goal.

B. Domain ε design

Recovering a domain description from source code is a typical reverse engineering task [7]. However, typical reverse engineering tools produce design models, not domain models [7]. One the one hand, classes that help in realizing non-functional requirements such as modifiability, reusability, maintainability, etc. are part of a design model. This introduces terms such as “visitor” and “observer” [7]. One the other hand these implementations also refer to technical domain concepts, introducing for
example “database”, “connection”, “file”, and “button”. The result of applying a design reverse engineering method thus results in a domain model that is perhaps complete, but neither consistent, nor concise. A design model is full of arbitrary technical overhead, from the perspective of domain knowledge recovery.

Domain knowledge recovery can be seen as design recovery with an additional filtering step. This filtering step removes all identifiers that are related to the technical implementation level and not the domain level.

C. Formal Concept Analysis

In this work we use formal concept analysis (FCA) as a first step to recover and study where and how domain concept identifiers occur in source code.

FCA is often used analysis method in reverse engineering tasks [1], [2] and other software engineering activities [3]. It is a data analysis technique that relates objects to their attributes (properties). A concept is defined to be a set of objects, of which each element shares the specific set of attributes. For example an object could be a class name ‘A’, and an attribute could be ‘calls a method from class B’. Intuitively, the idea is that group of objects that all have the same properties should conceptually be instances of the same kind. So, all classes that call methods from class B and no methods from other classes would form a concept.

Formal concept analysis takes a table of concepts and their attributes (a relation) as inputs and produces a lattice of concepts (Figure 1). It is a lattice because concepts are partially ordered using the subset inclusion relation between the sets of objects of each concept. Each lattice produced by FCA has the concept with the minimal set of objects and the full set of attributes at the bottom, and the concept with the full set of objects and the minimal set of attributes at the top.

D. Contributions and Roadmap

The contributions on this paper are:

- Automatically ranking the concepts from FCA lattices to recover domain identifiers (Section II)
- An experiment on 20 systems for the same domain, producing weights and thresholds for lattice ranking schemes (??).
- A validating experiment on two medium sized software projects showing promising results (??).

We summarize our conclusion in ??.

E. Related work

Some research has been done on domain knowledge recovery, often under different names. Diaz et al. called this Automatic Domain Analysis (ADA) [4]. They used all available artifacts for the analysis besides source code. Their approach requires human assistance for deciding if the recovered concepts indeed belong to the domain. Identified domain concepts are then captured in the relationship representation model RSHP and connected to the source artifacts (like UML models, textual specifications etc.). The focus is on the relation between concepts and software artifacts.

Similarly focused on the relation between domain and source code, Li et al. use the source code as input for their domain knowledge recovery approach [5], next to other input from a “domain knowledge base”. Their approach then finds matches – based on defined recovery rules – between the source code and the domain knowledge base. This works of course only if the domain knowledge has previously been defined.

FCA has been used successfully for different software engineering activities, a summary of these can be found in [6]. The ones most relevant for this work were used for the purpose of software maintenance, and here primarily for identifying modules by grouping conceptually related objects [7], [8], [9], [10], [11].

When examining concept lattices, we interpret them based on the positions of and relations between the contained concepts and the earlier described properties of these concepts. Caprile and Tonella e.g. used FCA for identifying general parts in function identifiers by using the identifiers as objects and the identifier parts as attributes, so that the most general parts are moving up in the resulting lattice [7]. If applied for domain knowledge recovery, these properties could also be helpful for finding possible domain candidates.

All efforts reported in the FCA literature do require human assistance for the interpretation of the lattices, and often also human improvement of the diagrams is required.
in order to be satisfactory [?], [?]. In the current paper we add an automated interpretation step after FCA.

II. USING FCA FOR DOMAIN RECOVERY

A. Overview

Our approach to domain knowledge recovery has two stages, as depicted in Figure 2 and Figure 3:

1) We have taken the first 10 of 22 different student implementations of a system called RICHRAIL in the first experiment. We have an accurate and simple reference model of the domain of RICHRAIL.

2) From these implementations we extract facts to use as input contexts for FCA. We experimented with 6 different types of contexts extracted from source code of Java systems.

3) Then we compute concept lattices for each pair of implementation ∗ kind of relation.

4) Finally we apply 6 different ranking heuristic to the resulting concept lattices and compare the result to the reference model to measure their accuracy. Each extracted relation is paired with a unique ranking heuristic.

This initial experiment (Figure 2) produces evidence that the approach might work, as well as a comparative analysis between the 6 types of contexts used as input and the used heuristics for filtering.

In Stage two (Figure 3) we combined the 6 ranking schemes, applying them to the other 10 RICHRAIL systems, to JHOTDRAW and to SYNPOS. The latter was used by Larman [?] as a case study in domain modeling such that we have an accurate reference model. JHOTDRAW is a well known benchmark for which we assume the domain model is known.

The combination of the approaches entails assigning relative weights and thresholds to each ranking, such that simple ranking is acquired. We use the results from Stage one to judiciously choose these weights and thresholds. We then run the resulting complex ranking scheme and compare the results to the reference models again to measure its accuracy.

B. Where to look for domain concepts?

Our basic assumption is that domain concept identifiers are mainly to be found in (implicit) data representations. The classes that represent data can be hidden anywhere in a system, yet they might have a typical shape or a typical way of interacting with their surroundings.

One hypothesis is that domain concepts are “bottom” classes, i.e. do not use much other functionality other than generic containers. They are in a data representation layer [?]. This layer may or may not be separated in a package. If we can identify the member classes of this layer, then we may have found our domain concepts. For example, data object may be expected to have high fan-in and low fan-out in a class dependency graph.

Another hypothesis is that domain concepts often participate in particular design patterns, such as “Factory” and “Command” [?]. The automatic detection of design patterns was subject to research by e.g. [?], [?]. We do not directly use pattern detection in the current paper. Nevertheless, we can expect that concept identifiers occur as parts of class names that implement design patterns. Therefore we will definitely have to tokenize class names.

C. Evaluating accuracy

TODO: Jurgen: rewrite and add manual synonym resolution, definition of recall and precision.

All findings where the name did not match with the expected domain model were examined manually. We counted findings as match with an expected result if the meaning in the domain of the students’ project was equivalent to our expectations. In some cases the domain of the students’ project was more detailed than expected, like e.g. a distinction between a locomotive and a wagon, which are both of type rolling stock. We counted these concepts also as positives.

As discussed earlier, we argue that a high precision is more important for domain knowledge recovery than a high recall. It is easier to look for additional conceptual classes manually if one knows reliably that recovered concepts indeed belong to the domain. It would be harder to distinguish between domain parts and technical if both are mixed up in the result, even if we know that most of the concepts are included in the result (the situation with a high recall and a lower precision).

D. FCA context creation and tokenization

An FCA context is a table with object identifiers labeling each row and attribute identifiers labeling each column. Table I summarizes the kinds of contexts we extract from Java code. Each context created is motivated from the particular ranking scheme (see Section II-E). C1 and C2 relate the name of a class declaration to the names of the field members declared in these classes and their superclasses. C3 and C4 relate the name of a class declaration to the names of all classes used in methods and fields of these classes. C5 related the names of classes to the names of their methods and names of the methods of their superclasses.

We extract three kinds of identifiers: full, tokenized and the powerset of tokens. The full identifier is the literal id as it occurs in the source code.
Fig. 2. Overview of Stage one, including 6 automatic ranking schemes of concepts in concept lattices.

<table>
<thead>
<tr>
<th>Id</th>
<th>Objects</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>class (full)</td>
<td>x field member (full)</td>
</tr>
<tr>
<td>C2</td>
<td>class (powerset)</td>
<td>x field member (full)</td>
</tr>
<tr>
<td>C3</td>
<td>class (full)</td>
<td>x use of class (full)</td>
</tr>
<tr>
<td>C4</td>
<td>class (full)</td>
<td>x use of class (tokens)</td>
</tr>
<tr>
<td>C5</td>
<td>class (powerset)</td>
<td>x declared methods (tokens)</td>
</tr>
<tr>
<td>C6</td>
<td>package (full)</td>
<td>x called packages (full)</td>
</tr>
</tbody>
</table>

TABLE I
FCA CONTEXTS EXTRACTED FROM JAVA SOURCE.

(a) original context

<table>
<thead>
<tr>
<th>Staton</th>
<th>AddWagonToTrain</th>
<th>drawWagon</th>
<th>drawTrain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Img_Display</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

(b) context using tokenized attribute names

<table>
<thead>
<tr>
<th>Staton</th>
<th>Add</th>
<th>Wagon</th>
<th>To</th>
<th>Train</th>
<th>Draw</th>
</tr>
</thead>
<tbody>
<tr>
<td>Img_Display</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Fig. 4. An example of using tokenized attribute names in the context.

The tokenized identifier produces a set of smaller identifiers by splitting them using “camelCase” notation and common separators (like ‘_’) as criterion. Each of these tokens is canonicalized by capitalization and stemming (using Porter’s algorithm [?]). Smaller tokens are motivated by the use of design patterns (??), and other forms of identifier composition commonly used by programmers [?]. An example of this is given in Figure 4.

Finally, the “power set” of a tokenized identifier is computed, taking the power set of each set of tokenized, capitalized and stemmed parts of a single identifier and then concatenating the elements of each sub-set — in their original order. The motivation is that a particular consecutive pair of tokens may proof to be a stronger concept than any arbitrarily ordered sub-set of tokens. An example is given in Figure 5 which also shows that the application of powerset of name results in the fact that the domain concept Wagon now has three attributes (marked in blue in the example), one more than the other objects.

E. Ranking the output of FCA

Each run of the FCA algorithm produces a lattice. In this lattice each concept has a particular rank, given a direction — either the top node ranks 0 or the bottom node ranks 0. A ranking scheme picks a direction and a range \( n - m \), then selects the \( n \)th to the \( m \)th ranking concepts in a concept lattice, possibly applies another heuristic filter and finally enumerates either their respective objects or attributes in an ordered list. We have experimented with 6 different ranking schemes.

a) Scheme 1: employs the C1 and C2 contexts, mapping class names to field names. We expect that domain identifiers happen to be referenced and stored by many different classes (high fan-in). In the resulting concept lattice we expect that objects with domain identifiers flow to the bottom because they should occur in relation to many different attributes. This should be stronger for the power set, since that allows an object identifier to occur for more different class declarations.

b) Scheme 2: similarly builds on high fan-in. We use C3 and C4, which relates a class to the classes it uses in field, variable and parameter declarations. Class names are now both objects and attributes. We expect domain identifiers to be attributes to many different objects (fan-in). So, we should expect that concepts containing domain concept attributes flow up in the concept lattice.

So, by ranking top-down and selecting the bottom two ranks we may hope to have identified some of the domain concepts.

b) Scheme 2: similarly builds on high fan-in. We use C3 and C4, which relates a class to the classes it uses in field, variable and parameter declarations. Class names are now both objects and attributes. We expect domain identifiers to be attributes to many different objects (fan-in). So, we should expect that concepts containing domain concept attributes flow up in the concept lattice.

Naturally, also utility classes will flow up and highly general identifiers for common design patterns. This would almost certainly lead to many false positives. We propose to select only rank 1 (top-down) and then filter
the set of attributes from this rank as follows:
\[ \{a_2|C_1, C_2 \in \text{rank}(1), a_1 \in C_1\text{.intent}, a_2 \in C_2\text{.intent}, a_3 \in C_2\text{.extent}, a_2 \notin C_1\text{.extent}\}. \]

Thereby we select only the attributes that are not part of the objects in some of the other concepts. TODO: Christian, what is the rationale behind this?

TODO: Jurgen: rewrite 3 to 6

c) Scheme 3: As context for this scheme we again use classes as objects and all declared types per class as related attributes. We then also add all parts of the declared type identifiers as related attributes, which where obtained by tokenization. This merges the ideas of schemes 1 and 2: conceptual domain classes will be more often used by other classes and domain concepts can be found in parts of other class names (the declared types) as well.

After the application of FCA we sort the lattice using the downrank strategy. Then we use the attributes of the concepts located in the upper half of the lattice as domain candidates. The weights assigned to them use 0.5 as the highest weight for the highest rank. The other weights are assigned using steps of (0.5/rankCount).

d) Scheme 4: The hypothesis for this step is that domain concepts not only appear as parts of other class names too, but also as parts of method names. We therefore construct a context in this and the following scheme which reflects this property. As objects we use the identifier powersets of the classes. As attributes, all tokenized method names of the methods declared by the original class are used and related to the corresponding classes.

In scheme 4 we identify the domain concepts as attributes which are close to the top. The lattice is sorted using downrank and weights are assigned to the attributes using 0.5 as the highest weight for the highest rank. The other ranks are assigned using steps of (0.5/rankCount) and all but the lowest ranked concept are considered as observation of test results showed that this lowest ranked concepts did in general not contain domain concepts.

e) Scheme 5: Scheme 5 implements the second part of the idea of scheme 4. We still sort the lattice using downrank, but reverse the ranks so that the highest ranks are at the bottom of the lattice. The objects of the concept extents are considered as domain candidates, and the weights are assigned to them using steps of (0.5/rankCount) and 0.5 as weight for the highest rank. All concepts are considered.

f) Scheme 6: This step tries to find packages which probably contain conceptual classes and makes use of the earlier described idea that often domain knowledge is grouped into one or more packages. We use a package call graph as input for the context where packages are the objects and the packages called by them are the attributes. After constructing the lattice, the same algorithm as in scheme 2 is used, but this time at package level.

After identifying possible domain packages, all classes contained in these packages are determined using fact extraction (see also the dashed line in Figure ??) and are assigned a weight of 0.4. The list of these classes plus the weight is the result of this last scheme.

This scheme is only used if the package call graph is not empty.

F. Implementation details

We use the Rascal meta programming language [?] to extract facts from Java projects in Eclipse and to form relations that can be input to FCA. T

The FCA algorithm is included in the standard library for Rascal\(^1\). The Java front-end for Rascal extracts all its information from the Eclipse JDT\(^2\).

Finally the output of FCA is a relation again which is processed in Rascal using relational calculus operations to obtain the final ranking.

We use Rascal’s Figure library, and the R language\(^3\) for visualizing and analyzing our results.

III. STAGE ONE: RICHRAIL EXPERIMENT

We applied 6 different ranking schemes to 12 different implementations of the same system called RICHRAIL. We summarize the results here, with the goal of analyzing the accuracy of each ranking scheme and producing thresholds and weights for a combined ranking scheme to be used in Stage 2.

Ranking Schemes: Examination of the results of the different ranking schemes shows that ranking scheme 2 on average gives the best results, which is also the reason why the a maximum weight of 0.6 is given in this scheme. Ranking schemes 3, 4, and 5 — with a maximum weight of 0.5 — give qualitatively equal results, even if differently distributed per RichRail project. Ranking schemes 1 and 6 show a diversification in the quality of the results, so that the lowest maximum weight of 0.4 is used for these.

It can be observed that ranking schemes, if applied in isolation, only exceptionally will give the desired results. A combination of all ranking schemes in contrast always improves the results.

Thresholds and weights: HERE WE SHOULD MOTIVATE THRESHOLDS AND WEIGHTS FOR THE NEXT VALIDATING EXPERIMENT

IV. VALIDATING A COMBINED AUTOMATIC RANKING SCHEME

TODO: intro with validation on the rest of the richrail projects and jhodraw and stuff

A. Results from RichRail Projects

B. Results

The average precision and recall over all RichRail projects are shown in Table [III] As can be seen, there is a small difference in the precision average, which will be discussed in more detail in the next section. There is no significant difference in the recall between the results for the projects used during development of the method and the reference projects.

\(^1\) Implemented by Bert Lisser.

\(^2\) http://www.eclipse.org/jdt

\(^3\) Say R!
Figure 6 shows a class diagram of one of the results with 100% precision and recall. These were conceptually identical and only have minor differences in naming, as not all student groups used the same names and/or naming conventions.

We stated earlier that the implementations of the students’ projects varied widely. RichRail16 and RichRail17 e.g. did only use the default package and 11 respectively 7 different classes (without any design patterns) to implement the whole system. Other implementations included different design patterns like Observer, Command, Interpreter, or Facade.

Discussion of outliers

As can be seen in Table II, the results include two outliers in terms of precision – RichRail10 and RichRail22 – which can give indications for possible improvements of our method. We therefore examine these outliers in the next sections.

RichRail10: The concept GraphicDisplay obviously does not belong to the domain. In our method we implemented a simple approach for removing technical concepts, where also the keyword “display” was used for non-domain concept exclusion. A more solid identification of technical parts – based on common technical concepts or the roles concepts play in design patterns – should also identify this concept as a technical concept and exclude it from the result.

RichRail22: The main reason for the low precision for the results of RichRail22 is that the students used the dutch language for naming the identifiers. The two concepts Weergave and ProRailWeergave can be translated as Display and ProRailDisplay, these concepts could also be automatically detected as discussed in the section on RichRail10.

The two concepts Mutatie and MutatieRegel are used to log the changes (mutations) of the domain. They were also located in the model package, which contains the most parts of the domain in this project. As these two classes are technical, but do not belong to a (known) design pattern, further research is necessary to ensure that also concepts like these are filtered out of the result of our method.

The concept Controleur is an abstraction of actually three controllers, which serve as facades on different levels.

C. Result from JHotDraw

Riehle documented in his dissertation some central abstractions of the JHotDraw framework [7], based on the JHotDraw tutorial and shown in Figure 6. As these can be seen as part of the domain model from JHotDraw, we can use them as reference for determining the effectiveness of applicability on larger systems.

The analysis was executed using JHotDraw version 6. Table IV gives the probable conceptual classes which were recovered from JHotDraw. The classes marked in bold are also part of Riehle’s summary of the main abstractions of JHotDraw. The classes marked in italic are identified by the author, after manual examination of the related source code, as probable part of the graphics domain.

While precision and recall can not be given here because no original domain description exists, this result still
indicates that the method delivers good results. We think that the recovered concepts belong for a high percentage to the domain.

Even if it is harder to find a appropriate threshold here, the result is still useful for domain knowledge recovery with human assistance, as it provides a ranked list of domain classes. This makes it easier for a manual setting of the threshold: just stop traversing the list downwards when the amount of obviously technical concepts becomes to big.

**Ranking Schemes:** Examination of the results per ranking scheme shows that for SynPos the ranking schemes 1, 3, 5, and 6 give the best results.

**Conclusion:** We think that this case study indicates that the DoKRe method is also applicable for medium sized systems, even though no precision and recall could be calculated. The recovered concepts belong for a large part to the graphics domain. However, in order to make general conclusions, a more grounded validation should be executed which uses a well described domain model as reference.

**D. Result from SynPOS**

Larman uses in [?] a point-of-sale (POS) system as case study for domain modelling. We can use the basic domain model as described by Larman to compare them with a POS system. This model is shown in Figure 7.

The case study in Larman’s book was fictional, so we assume that CreditCard is also part of the domain, but it is not included in the domain model as described by Larman [?]. This leaves four non-domain concepts: SwingWorker, Synchronizer, StoreDB, and I18N. The precision for the analysis of the SynPOS system is therefore 64.6%. The recall will not be calculated, as Larman’s domain model is not complete and seems to differ in some points from the implicit domain model in SynPOS. Source code examination reveals e.g. that SynPOS does not make a distinction between the conceptual classes Manager and Cashier, but realizes this as attribute of the conceptual class User. Also SynPOS is intended for use in one store, and therefore does not include the conceptual class Store.

Examination of the non-domain concepts in the source code leads to the following results: SwingWorker is a threading mechanism, used for executing actions in an apart thread. Synchronizer manages the actions which need to be executed and controls the SwingWorker. StoreDB is a database wrapper, used to execute SQL-statements on a database. I18N is used for localization support and handles resource bundles.

These results can be used for further improvement of the automatic detection of technical concepts.

**Ranking Schemes:** Examination of the results per ranking scheme shows that for SynPos the ranking schemes 1, 2, and 5 give the best results.

**Conclusion:** The recovered domain concepts from SynPos presented in the final result, with a precision of 64.6%, indicate that the method needs improvement. However, the non-domain concepts recovered give useful starting points for this improvement, as they suggest where to focus on when looking at the technical concepts.

**V. CONCLUSION**

The results presented in the previous chapter show that our method produces good results for small programs with a high precision of the recovered domain concepts in the form of conceptual classes. If applied on larger systems, the results are promising, but need further improvement.

**A. Evaluation of the method**

1) **Threshold Definition:** Defining a reasonable threshold is essential for the quality of the results of our method.
We used a threshold of 1.5 in our experiments. This value was based on earlier observations using the RichRail projects and also applied for the analysis of JHotDraw and SynPOS.

However, the definition of a threshold includes a trade-off: increasing the threshold leads to a higher precision in most cases, but also to a lower recall.

If we use a threshold of a total weight of higher than 1.8 for the analysis of the RichRail projects, then we get an average of 97.27% precision (with only one result left having false positives), but only 62.5% recall. If we look at the results of JHotDraw in Figure 4, then we clearly can see that a threshold of 2.0 would lead to 100% precision (if Larman’s domain model is used as basis), but would also decrease the recall. Using for SynPOS a threshold of more than 1.5 (instead of more than or equal to) would lead to a precision of 77.8%, an increase of 13.2%. The recall would not change in this case.

Research on how a threshold could best be defined or probably automatically determined, based on properties of the system to be analysed, should be subject to future work.

2) Domain Knowledge Properties in Source Code: In general, the method just uses a small subset of possible applications of the intuition assumptions on properties of domain knowledge in source code. This subset proves that it is worth to explore these ideas in more detail in order to improve the method.

If the assumption in ranking scheme 2 only seems to work for small systems, then one could probably try to find modules first using existing approaches and then apply this technique to determine possible domain concept candidates. This is not implemented in our approach.

B. Threats to Validity

Good and consistent naming of source code artifacts is an essential quality aspect of software systems. Without
proper naming it is hard to understand the purpose of code fragments and also the semantic meaning of them. This also counts for the presented technique: without a proper naming of the source code artifacts the technique still will identify domain concepts, but these probably can’t be interpreted. The technique is also not able to handle the usage of abbreviations in combination with complete names, e.g. researcher and researchr are identified as different concepts and, because they both get lower weights, won’t be identified as domain concepts either. Maybe LDA can be used in future work to solve this problem, as LDA can handle synonymy.

At this moment the technique can only be applied to Java-projects in the Eclipse IDE. In order to make it usable for other programming languages as well, Rascal has to offer fact extraction first for these other languages.

Due to the used methods (like tokenizing identifiers and using power sets of their name parts) and the combination with lattices, the performance of the system is not sufficient for large systems at this moment.

It is not known if different domains can have different properties in relation with source code, which might be hypothetically possible. We try to prevent this by using three different domains in the case studies, but can not claim that the identified properties are valid for all possible domains.

The reference implementation of the DoKRe method in Rascal is at this moment only able to recover domain concepts from Java projects. However, other object-oriented languages include programming constructs and language elements different from the Java ones. We did not evaluate if the domain knowledge has the same properties when these constructs and elements are used.

There exist many different coding styles and naming conventions. The method tries to take the most common ones into account, but there probably are some styles and conventions which might lead to other results.

We did not evaluate the effect of design decisions and architectural patterns on the method. The RichRail case studies offer a broad variety of designs and prove that these are of less impact on the results, but this can not be generalized for larger systems.

We assume that all packages and classes of a project indeed belong to the implementation. If frameworks are used, then it is common that application parts and framework parts are highly intermingled, making a distinction between these two more difficult. But without a distinction there is a higher chance that also concepts contained in the framework will be recovered, which is not always desirable.

In order to prevent overfitting, we used only 12 of the RichRail projects for the evaluation of the method and the other 10 projects for validation. However, all RichRail projects are small and contain a small and simple domain, due to the limitations of such projects in an educational context — mainly time and complexity. There is a chance that the method works better for this small domain than for other possible small domains.

The maximum weights for the results of the specific ranking schemes are determined by observing the results using the first 12 RichRail projects. As the case studies of JHotDraw and SynPos show, does the effectiveness of the ranking schemes differ per system. We did not examine if there is a relation between some properties of a system and an optimal weight distribution between all ranking schemes for this system. It is therefore possible that the static distribution used in this work does not give the best possible results for other systems.

The interpretation of the results from the larger systems is mainly based on the manual assessment by the author. This could lead to a biased view during the assessment. To prevent this we use reference models for the evaluation of the results from RichRail (provided by the author) and SynPos (from [7]), but no such sufficient reference model exists for JHotDraw.

The removal of technical concepts is implemented in a straightforward approach, based on naming only. If other names are used for the same technical concepts, than the method in its current state will fail to identify them properly. As suggested in future work, this has to be improved.

C. Final conclusion

In this work we show that it is possible to describe properties of domain knowledge in relation with source code, and that these properties can be used for the definition of the basic automated domain knowledge recovery method DoKRe. The method makes use of Formal Concept Analysis, and implements the automated ranking and interpretation of concept lattices.

The method, supported by the promising results from the experiments, adds to the field of domain knowledge recovery some new ideas for minimizing the amount of human assistance needed. The experiments show that the DoKRe method in its current state already leads to good results with small systems and promising results with medium sized system. We define and use hereby only a small subset of the possible ranking schemes and future work, exploring these other possibilities, should lead to further improvement of the method.