DoKRe - A Method for Automated Domain Knowledge Recovery from Source Code

Christian Köppe

30 August 2011

Master Software Engineering

Supervisor: Jurgen Vinju

Organization: Centrum voor Wiskunde en Informatica

Publicaton Status: public

University of Amsterdam

Contents

Abstract

All software systems contain knowledge about the domain they are used in, but this knowledge is for a great deal latent and therefore not explicitly available. Even if the domain is documented, this documentation is often incomplete or outdated. But having a representation of the domain of a system can be valuable and sometimes even necessary, as e.g. for the design of Domain Specific Languages (DSL) or the re-engineering of legacy systems.

We first explore the possibilities of the automation of domain knowledge recovery. Based on these possibilities the DoKRe method is defined as basic approach, which makes use of Formal Concept Analysis. We show with a prototypical implementation of the method and a number of experiments that our approach gives promising results and can be used as starting point for further research on automated domain knowledge recovery.

Preface

The two years of my part time study of the master software engineering at the University of Amsterdam were two of the most intense years in my life, but also two of the most enriching ones. I entered the wonderful world of academia and filled a gap in the theoretical part of my software engineering knowledge. For this I am thankful to all lecturers of this master.

The most important was Jurgen Vinju, as he also supervised my thesis and helped me to make this a piece of work where I am proud of. I very much enjoyed our inspiring discussions and I hope we have more of these in possible future joint projects or in some other ways.

Then I want to thank Hans Dekkers. Even if I have to say that it was not always easy being his student, he definitely pushed me and the other students to reach higher levels of thinking and, especially, asking the right questions.

I also learned a lot from Paul Griffioen, and especially his general openness made all conversations and discussions with him rememberable moments.

Furthermore I want to thank the other lecturers from this master as well: Paul Klint, Tijs van der Storm, Jan van Eijck, and Hans van Vliet.

I appreciate that I was able to write this master thesis at the Centrum voor Wiskunde en Informatica (CWI), as the people and the environment there — especially the software engineering group lead by Paul Klint — are highly inspiring and I enjoyed the open and creative atmosphere there a lot. A special thanks goes to Michael Godfrey for some last minute, but highly valuable, feedback.

I had a couple of exciting fellow students with whom I had the privilege to work together in projects. Beside a pleasure to work with, Jouke Stoel was also an important sparring partner during the writing phase of my thesis. We exchanged many ideas, so I also want to thank him for the inspiration. Also important were (and still are) Davy Meers and Arie van der Veek. If all publications could be written as easily as our collaborative one...

And, to save the most important person until last, I am very thankful to my partner Anne. I would not have managed this master without her! I am glad that I can give it back now...

*The method name DoKRe is an acronym of **Do**main **K**nowledge **Re**covery. It is pronounced similar to Ogre, the name of the species where also Shrek[1] belongs to. Shrek himself describes ogres as onions, consisting of layers where you do not see the inner beauty because of the outer layers. Our method DoKRe also makes use of this metaphor. It pulls off the outer layers of a system in order to reveal the beauty inside: the domain! In that sense, the similarity in spelling is not that coincidental...*

[1] Steig, William *Shrek*. Farrar, Straus and Giroux, US, 1990

Chapter 1

Introduction

The analysis of software systems is an established research area. The goals of these analyses differ widely, but it seems that more authors put the focus on the technical aspects of the system — the *how* — in order to reveal elements and structure of the technical implementation. This is not sufficient for activities like system re-engineering or DSL engineering. Here the interest is mainly in the domain of a system — the *what*.

1.1 Importance of Domain Knowledge

A domain is the subject area of the activity or business of a user. All information that is relevant for this domain constitutes the knowledge of the domain.

All software systems serve purposes of one or more special domains. These domains are usually analysed and described during the requirements phase of the development process, using textual specifications or notations like UML. This domain knowledge consists of noteworthy concepts representing the domain. The domain analysis results then can act as inspiration for designing software objects [31], which eventually will be part of the source code of the system. This source code also reflects the knowledge of the domains as this is necessary for offering the functionality needed to serve the domain purposes.

Knowing these domain concepts is important for different activities. Program understanding, a part of maintenance, is probably the most important one. De Souza et al. performed a survey under maintainers about the importance of artifacts for the maintenance process. More than 85% of them state that a conceptual data model[1] is important or very important [16].

Domain knowledge also plays an important role for re-engineering or reverse engineering of old systems. It can be of value for business analysts too, as long running systems implicitly contain knowledge about the *real* elements and structures of a business domain. These can differ from the concepts described earlier in the software's lifecycle — during requirements gathering — as the systems reflect also changes which were based on new or adapted requirements and which were not incorporated into the initial domain specifications.

The domain model can play an important role in many aspects: if we want to start from scratch, getting return on investing years in fine-tuning the system to the business' needs, perhaps re-engineering into a domain specific language (DSL), for which a domain model is the first step, or as input for requirements engineering of a next generation software product.

1.1.1 Why is Domain Knowledge hard to obtain?

Typical reverse engineering results in different design models of the system using some graphical notations like UML. However, design models are different from domain models as they always take also the technical aspects of a software system into account. Larman also makes a clear distinction between pure domain models and design models [31].

Classes and objects that help in realizing non-functional requirements such as modifiability, reusability, maintainability, etc. are added to the system design. A traditional reverse engineering of a system written in an object-oriented language would therefore not lead to a 'clean' domain model, but would be interspersed with technical elements and concepts. This makes it hard to use well-known reverse engineering strategies for the recovery of pure domain knowledge.

Growing systems usually also contain a growing amount of domain knowledge, and naively one could assume that the difficulty level of extracting this domain knowledge roughly stays the same. The extraction of this

[1] A conceptual data model can be seen as equivalent to the domain model and is differentiated from logical and physical data models.

domain knowledge in the early phase of the system lifecycle is relatively easy, as the domain concepts can usually be more easily identified in the structure of the system's architecture. But over time the intermingling between domain and technical parts probably will increase due to software entropy, which makes it harder to extract the domain knowledge. We call this the *Domain Knowledge Recovery Paradox*[2]:

> The older a software system, the more domain knowledge it implicitly includes, but also the more difficult it becomes to recover this knowledge.

1.1.2 Sources of Domain Knowledge

Growing systems are subject to software entropy unless work is done specifically to reduce the complexity, as described by Lehman's second law [20]. But not only the structure of the system evolves, also the documentation of the system gets more out-of-date or is not present at all [48]. Maintainers and reverse engineers have to get their information out of the available sources. This is often the source code only, as documentation of the adapted domain does not evolve the same way as the source code and the humans who have this knowledge are not (anymore) available. Even if the domain adaptations are documented, it is likely that this documentation is scattered through different models, issue-trackers, comments and other artifacts.

Some authors state that the grade of documentation and source code being "out-of-sync" increases with the lifetime of a system [22, 48]. So this documentation can not be used to *reliably* discover the real domain concepts latent in the software system. In agile development and the Unified Process, domain models are optional artifacts. It therefore might be possible — and is probably true for most projects — that no domain models exist at all.

We conclude that the source code is the artifact that contains the most up-to-date information about the domain in terms of accuracy and completeness. It reflects all changes related to the original domain description, as e.g. added or removed features, which might not be documented in other easily accessible artifacts. The source code is available for most systems and therefore provides a good source for recovering information about the domain if no other sufficient domain documentation is available.

1.2 Reverse Engineering and Information Retrieval

The technique most often used for the recovery of the design and other information of a system is *reverse engineering*. It is defined as "the process of analyzing a subject system to (1) identify the system's components and interrelationships and (2) create representations of the system in another form or at a higher level of abstraction" [13]. Much research has been done on reverse engineering from software developed with both object-oriented and non-object-oriented technologies. However, as object-oriented systems are the legacy systems of tomorrow [48], it is important to scrutinize the possibilities and problems related to recovery of domain knowledge from these systems.

The recovery of domain knowledge can be seen as a special branch of reverse engineering. The desired results, the concepts of the problem domain, are on a much higher abstraction level than the mostly technical results of other reverse engineering branches like design recovery or aspect mining. The following definition of domain knowledge recovery is discussed in Section 3.2 in more detail:

> *Domain Knowledge Recovery* (DKR) is the discipline of making the domain knowledge, which is implicitly contained in an existing software system, explicitly available in the form of a domain description.

However, the two essential steps — fact extraction and analysis of these facts — can be found in all branches of reverse engineering. Two known analysis techniques applied in software engineering research are Formal Concept Analysis (FCA) and Latent Dirichlet Allocation (LDA). FCA has a long history in information science, and different groups of researchers used it also for software engineering purposes. LDA is relatively new and was first introduced in 2003 by Blei et al. [7].

Many studies proved the general applicability of both techniques for the analysis of different kinds of information. However, all of these studies were performed from a technical point of view, focussing on information about technical aspects of the systems. None of these studies discussed the applicability for the extraction and analysis of pure domain knowledge. One possible reason might be that there is no clear definition of what the extracted domain knowledge precisely consists of — there exists no metamodel describing the parts of a domain model. It is therefore difficult to compare which of the known techniques delivers better results and if they

[2]This term was created during a "sparring" session with my supervisor Jurgen Vinju.

could be adapted or combined in order to improve the results. Different authors give different definitions of how to describe the elements and structure of a domain.

However, the existing approaches also have a few other short-comings. All of them require human assistance, which makes the results dependent on the expertise of some domain experts. Some approaches use already existing domain descriptions to recover these from systems [32]. The point here is that the domain knowledge has to be known in advance in order to recover, which makes these approaches actually more an artifact-domain concept-connection mechanism. What would be helpful is a fully automated mechanism to recover domain knowledge from existing systems without human assistance and which is not based on an existing description of the domain to be recovered.

Diaz et al. use all available artifacts *beside source code* for building the domain representation [18]. They also need to evaluate the quality of the artifacts they use, e.g. UML diagrams, hierarchical charts, product manuals, structured requirements specifications, ER data models, SQL databases etc. As stated earlier, many of these artifacts will not reflect the current state of the system which could lead to some conflicting results. They state that identifying these conflicts needs to be done. Another problem with their approach is that it again requires domain experts for many steps (as defining what artifacts are to be used, based on their perception if these artifacts contain domain knowledge in acceptable quality). Actually, an approach that could extract the domain knowledge of a system without the need of domain experts is desirable, even if unlikely to be feasible. However, minimizing this need already would be of great value.

This thesis presents a possible approach for the minimization of the necessity of human assistance for the extraction of domain knowledge from software systems.

1.3 Research questions

In this thesis we ask the following research question: "How can domain knowledge recovery be automated using Formal Concept Analysis and Latent Dirichlet Allocation?". To answer this question we need to answer the following five subquestions:

1. How can domain knowledge be represented in a comparable way?
2. Which parts of domain knowledge can already be recovered using FCA and LDA?
3. Which modifications of these methods could lead to their improvement?
4. Are there observable properties of domain knowledge in source code that can be generalized?
5. How can the need for human assistance be minimized with the implementation of the identified method modifications?

The rest of this thesis is organized as follows: we first describe the research method and the discuss the chosen approach in chapter 2. Then we provide the necessary background information in chapter 3, followed by an extensive discussion of our method in chapter 4. The experiments executed with the method using different case studies are presented and analysed in chapter 5. We conclude this work with a discussion of the results and possible improvements and suggestions for future work in chapter 6.

Chapter 2

Research Method and Approach

The project has two parts: identifying the problem and developing a method for solving it. Different research methodologies are used for these parts. We describe our approach next as well as the case studies we use for evaluation of the developed method and finally summarize our contributions.

2.1 Chronology

For the first part of this project we used *exploratory* research methodology. As there was no clear problem definition in the beginning of this project, a *secondary* research was done by performing an extensive literature study. This offered insight in specific problems of the field of Domain Knowledge Recovery (DKR) as well as the applicability of Formal Concept Analysis (FCA) and Latent Dirichlet Allocation (LDA) for DKR. It also led to the definition of a domain metamodel, which is described in section 3.1.2. Beside the literature study we also explored the general applicability of both FCA and LDA with prototypical implementations. This helped in defining a specific problem definition and led to the exclusion of LDA from the remaining project.

The activities of this first part led to the answers of the two research subquestions: "How can domain knowledge consistently be represented?" (section 3.1.2) and "Which parts of domain knowledge can already be recovered using FCA and LDA?" (see sections 3.3.4 and 3.4.4).

The results of part one were then used as basis for the second part of this project, where *constructive* research was used as methodology. The goal was to "construct" a method that can be used for automatically recovering domain knowledge. We used the theoretical knowledge obtained in part one of this project to develop a framework for our method. Then the properties of domain knowledge were described and prototypically implemented in Rascal. This prototype was used to test our hypotheses (defined in the next section) and to validate our method. We primarily performed qualitative validation using different systems for our experiments, but also were able to quantitatively validate our method with a group of 22 similar projects.

With this second part the last three research subquestions were answered: "Which modifications of these methods could lead to their improvement?", "Are there observable properties of domain knowledge in source code that can be generalized?" and "How can the need for human assistance be minimized with the implementation of these modifications?" (see sections 3.3, 3.4, and chapter 4).

2.2 Approach

To answer the first subquestion — how domain knowledge can consistently be represented — we summarize domain knowledge definitions as found in the existing literature. Based on this summary we develop a domain metamodel that can be used for generic descriptions of domains. This model is then used to evaluate known applications of both FCA and LDA in software engineering as a frame of reference, using published case studies. The focus is on their applicability for domain knowledge recovery.

We then explore possibilities of improving FCA in order to automate the process of domain knowledge recovery. Possible improvements are implemented in Rascal and evaluated using different case studies for the experiments.

2.3 Case Studies

In order show the applicability of the developed method and to validate it, we use a group of smaller case studies and two larger ones.

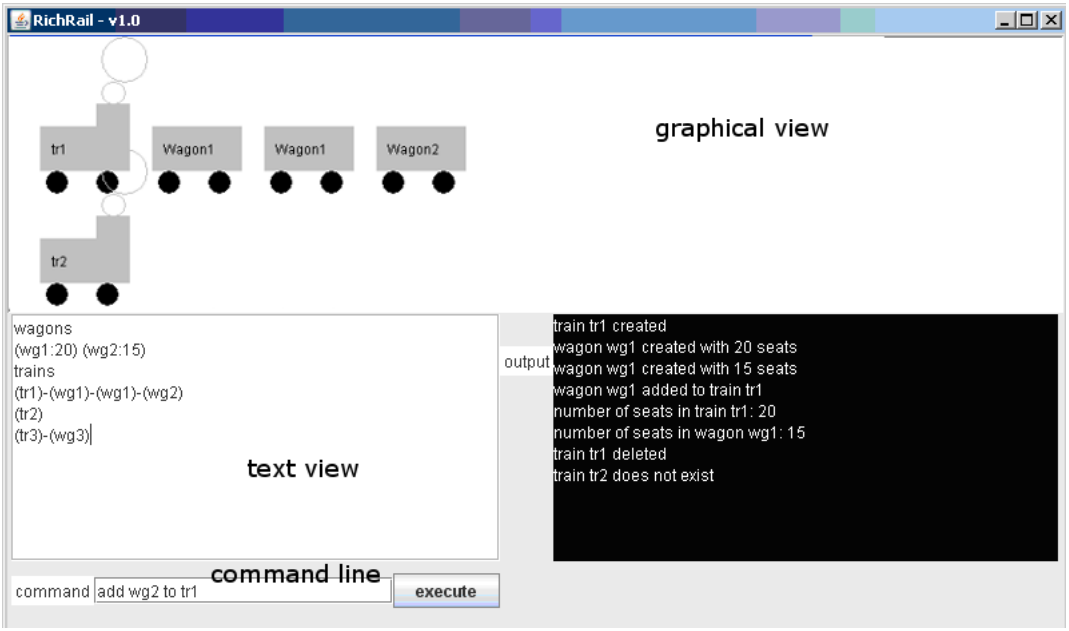


Figure 2.1: A screenshot from an exemplary RichRail user interface.

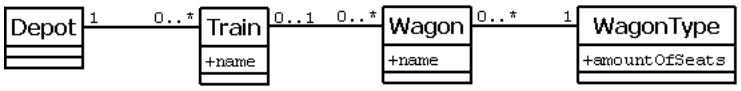


Figure 2.2: A possible domain model from the RichRail case studies (in UML)

RichRail

The smaller case studies are projects from 2nd and 3rd year undergraduate students of BSc Informatics/Software Engineering from the Hogeschool Utrecht / University of Applied Sciences and were part of a course on Patterns and Frameworks. These projects required the implementation of a small train administration, including different views and a small train administration DSL which can be used with a command line. Figure 2.1 shows a screenshot of an exemplary implementation. A longer description of the projects can be found in [30]. In total we have 22 different implementations of the RichRail system available.

The technical realization of the projects shows a great variety of implementation approaches, using e.g. different design patterns, package structures, or naming conventions. However, all students' implementations are expected to include a roughly identical trivial domain. These properties makes the projects suitable for validating our method, as it can be shown that it yields equivalent results independent of the technical implementation.

According to the description in the student assignment, all systems are expected to contain roughly the same domain concepts. This expected domain model is shown in Figure 2.2[1].

As stated earlier, the implementation of the RichRail systems differs widely, and so does also the size. The smallest system comprises 7 classes in one package. The largest system contains 38 classes and another large system defines 9 packages. The average size is about 17 classes and 4 packages.

JHotDraw

The first larger project is JHotDraw version 6 [3], which is a Java GUI framework for technical and structured graphics. Its domain covers graphical elements, their relations and ways of manipulating them.

Parts of the domain, the basic abstractions, are described by Riehle in [42]. These basic abstractions can be used to evaluate the method. However, as these abstractions are only a part of the domain of JHotDraw, the results are also evaluated based on the author's common sense knowledge of the graphics domain.

JHotDraw comprises 306 classes and 19 packages.

[1] Variations of this model are possible (it was not explicitly given in the assignment): class `WagonType` could be included as parameters `type` and `seats` in class `Wagon` and class `Station` could be implicitly implemented using a list.

SynPos

Larman uses a fictitious point-of-sale (POS) system as case study in [31]. According to him, "a POS system is a computerized application used (in part) to record sales and handle payments" [31]. During the case study, a larger description of the domain model for this POS system is developed. This can be used for evaluation of our method, as it can be expected that other POS systems also contain the basic domain concepts described in Larman's model.

The system we use is SynPos [6], an open source POS system. It basically contains most of the functionality also described by Larman, and therefore probably also most of the domain concepts. This makes SynPos an appropriate case study for the evaluation of our method.

SynPos is developed in Java and comprises 179 classes and 7 packages.

2.4 Contributions of this Work

We contribute three important steps to solving the problem of automated domain knowledge recovery from source code:

- A domain metamodel is defined (see section 3.1.2), which can be used as frame of reference for the definition of domain models and their evaluation and the evaluation of domain model recovery methods.
- Properties of domain knowledge are described and it is shown that these properties can be used to identify elements of the domain in an object-oriented system (see section 3.2).
- We show that it is possible to automate the process of domain knowledge recovery, at least for the recovery of basic domain elements.

We also provide experimental validation of the applicability of Formal Concept Analysis for domain knowledge recovery:

- We present ideas on how to automate the interpretation of lattices and show through a prototypical implementation that these work (see section 4.3). This can help in decreasing the need for human assistance for the application of FCA in general.
- We show, to our best knowledge for the first time, that Formal Concept Analysis can be successfully applied for Domain Knowledge Recovery (see chapter 5).

Chapter 3

Background and Related Work

In this chapter we first describe domain descriptions and define a domain metamodel. We then give a definition of domain knowledge recovery and address shortcomings in existing approaches, based on a review of the literature. Next, we provide necessary background information on Formal Concept Analysis and Latent Dirichlet Allocation and discuss their applicability for domain knowledge recovery. We finally present related work and discuss improvements.

3.1 Domain Descriptions

Unfortunately, it is hard to find a good definition in the literature of what *domain* means in software engineering. However, Evans describes a domain as "the activity or business of the user" and "the subject area to which the user applies the program" [21]. Examples of domains are business domains (banking, insurance, railroad administration, etc.), general domains (drawing, communication, etc.) or technical domains (databases, application servers, etc.).

A *domain description* consists of all meaningful concepts and abstractions of a domain, where *meaningful* is dependent on the viewpoint. For the banking domain these could be concepts like account, transaction, ATM, or customer; but also services like e.g. cash withdrawal. There could exist different descriptions of the same domain, again depending on the viewpoint. In banking, a customer does not need to know about employees and their provisions, whereas a manager would find these concepts probably very important. In this work we see domain descriptions from the software engineering viewpoint, meaning that all concepts of a domain — the non-technical concepts — are potentially interesting and therefore important.

Domain descriptions play an important role in software engineering, as they make the domain knowledge implicitly included in the domain explicit and visible. All developed software systems serve the purpose of specific or general domains. A thorough understanding of these domains is therefore essential for a successful realization of the software, and the key to a good understanding of the domain is its description.

Domains can be described in different ways: unstructured text or by using more formalized models. As unstructured text might be sufficient for small domains, it is not well suited for larger domains which also need to be validated and communicated. A reason for that is that such a text does not give a good overview and does not emphasize specific parts of the domain if needed. But unstructured text is the most common way to describe domains, even for large domains, and only in few projects a (visual) model is used.

3.1.1 Ontologies

An *ontology* in computer science is a specification of the concepts of a domain and the relationships between these concepts. Ontologies are a common way of describing domains, and many ontologies can be found describing all different kinds of domains. They seem to be less accepted in the industry, as the most non-academic books in the field of software engineering do not use them if it comes to domain descriptions. This is why we choose to not use the form of ontologies for our work.

However, because ontologies offer the possibility of modelling many different aspects of a domain, we use the description of the components of ontologies also as input for the definition of our domain metamodel[1]. Therefore our domain metamodel, defined in the next section, also offers the possibility to transform the domain model into an ontology if required.

[1] based on [29]

Part	Description	Imp.
Conceptual Classes (CC)	**What**: The "things" or key abstractions from the real world. **Why**: They form the core of a domain model and can be found in all domain model descriptions. Without them most other parts do not make sense.	1
CC: Attributes	**What**: The attributes of a conceptual class that represent its state. Properties are name and the attribute constraints: type (mostly mostly numbers or text), multiplicity, default value. Not included are attributes which represent associations. **Why**: Attributes are necessary in order to define the relevant properties of the state of a conceptual class.	2
Associations	**What**: The relations between conceptual classes that are not part of a hierarchy, optionally including the name and the association constraints: multiplicities, roles, direction, and type of association (knows, uses, is-part-of, etc.). **Why**: Conceptual classes do not exist in isolation, but are mostly interrelated with other conceptual classes. A domain model has to reflect this.	2
Inheritance Relations	**What**: The relations between conceptual classes that are part of a hierarchy: generalization and specialization. **Why**: Generalization and specialization are essential elements of human reasoning and can be found in all domains. As they differ essentially from associations, they are included as separate model part.	3
CC: Behavior	**What**: The responsibilities a conceptual class has, the events it can react on, and the possibilities to modify its state, focussed on what it can do and not how. **Why**: In some cases is the responsiveness of conceptual classes on external events – as part of their responsibilities or state modifications – essential for the domain, so it should be included in the model.	4
Services	**What**: Significant processes, transformations, or transactions where no conceptual classes are directly responsible for are modelled as stateless services. Services have no attributes. **Why**: There are domains where such stateless behavior plays an important role in the domain, it is therefore included. An example from the banking domain could be a funds transfer, which does not belong to a specific account and includes some business rules.	5
Modules	**What**: Modules are cohesive sets of domain concepts on a higher abstraction level. **Why**: Not mentioned often as part of a domain model, but modules can be helpful in larger domains to increase comprehensibility by introducing different abstraction levels.	6

Table 3.1: Domain Metamodel (including importance)

3.1.2 Definition of a Domain Metamodel

There is no formalized definition of what a domain model is and what domain models consist of. However, some descriptions can be found in the literature, e.g. in [21, 23, 29, 31, 32]. A summary of the domain models described in the literature can be found in Appendix A. We used this summary for identifying common parts of all described domain models as well as differences between these models. These identified parts formed the basis for the description of a domain metamodel which is shown in Table 3.1. For each part we describe **what** it represents and **why** it is included in the model. Also the importance of this part for domain descriptions is given, this importance is discussed in more detail in Appendix A.

3.1.3 Domain Model Representation

There are different possibilities of representing the domain model. However, the representation of the domain model is at this stage of less importance. Therefore we choose the Unified Modelling Language (UML) for the moment, as UML offers all possibilities of representing the recovered domain knowledge and is widely accepted.

3.2 Domain Knowledge Recovery

3.2.1 Definition

There is no standard definition of *domain knowledge recovery* (DKR), but there are few approaches describing this activity [18, 32]. Based on the common parts of all these approaches, we formulate a definition as follows:

> *Domain Knowledge Recovery* (DKR) is the discipline of making the domain knowledge, which is implicitly contained in an existing software system, explicitly available in the form of a domain description.

DKR is a branch of *reverse engineering*, as it recovers non-existing or incorrect artifacts – the domain model – based on existing artifacts, like the source code of a software system. While other branches of reverse

engineering are more focussed on the understanding of the internal working and structure of a system, DKR aims to extract only the domain concepts independent of the implementation.

Putting the focus on existing software systems distinguishes DKR from the activities related to *requirements engineering*, a forward engineering activity which usually takes place in the early phases of the software development lifecycle. Still, its output can be used for requirements engineering of a next generation software product.

3.2.2 Input

The input to DKR differs in all known applications and is dependent on the goal. Some approaches use all available artifacts, such as documentation, existing models, database schemas etc. [18]. Other approaches make use of the source code of a system [32].

If the source code of a system is used as input, or has to be used because it is the only available source artifact, then it is helpful to explore special characteristics of domain knowledge which is contained in the source code.

Software is often structured using the architectural LAYER pattern [8]. When layering is applied, the domain knowledge is probably located in the domain layer of a system. Even if no layering is applied is the domain knowledge often grouped in specific components or packages, which is supposed to improve reusability of the domain model. This property should be used for DKR, but it is certainly not sufficient to only look at this.

According to the rules of layering and encapsulation, domain elements (either layers or packages) are called by other elements, but do not call other elements themselves, which can be used for determining possible domain packages via a package call graph.

The domain knowledge is not limited to a potential architectural artifact like a domain layer or package, but can also be found in technical parts of the system. Therefore the whole systems' source code should be used as input for the analysis to recover the domain knowledge included in the technical parts as well. These can e.g. be parts of design pattern implementations. A typical COMMAND pattern implementation [24] could include classes like "Command", "Create**Wagon**Command", "Create**Train**Command", etc. So the domain concepts *Wagon* and *Train* are also included in the names of the pattern implementation classes.

Participants in a pattern implementation play specific roles, and sometimes these roles can be traced back to domain knowledge. The automatic detection of design patterns was subject to research by e.g. [28, 39]. These results could be used for determining the parts of (detected) design patterns which represent domain knowledge, based on the participant-roles as described by the Gang of Four in [24] and other pattern authors.

These properties can be used for domain knowledge recovery to calculate the chance for a concept of being a domain concept. Each of these properties can be implemented in separate steps for finding possible domain concepts, assigning weights for being a domain concept to the found results. At the end one can combine these weights to get a list of the most probable domain concepts.

3.2.3 Interpretation of DKR Results

All known approaches require human assistance, as their main goal was to relate artifacts to the parts of the domain which they contain and which was already described. These approaches are not feasible if no domain description is available a priori and they can not be applied in an automated way.

So DKR is especially interesting if no domain description is available at all or if the existing domain description is not sufficient or outdated. Acquiring such a domain description based on the existing artifacts with a minimum need of a domain expert for judging the results would also improve the applicability and the value such a method could have. A good DKR method should therefore be able to *reliably* recover the domain knowledge from an existing system with as little human assistance as possible.

Domain knowledge recovery methods can be validated using software systems where also the domain descriptions are available. The most usual measures for such validation are *precision* and *recall*. Precision means here the percentage of the amount of relevant found concepts in relation to the total amount of found concepts. Recall means the percentage of the amount of relevant found concepts in relation to the total amount of relevant concepts in the domain. So the method could then be applied for recovering the actual domain description from each system, which afterwards can be compared with the original one in order to determine the precision and recall of the method. Figure 3.1 shows this approach.

Both precision and recall can be considered as important, but especially for an automated approach we focus mainly on the precision of the method. Knowing reliably that the found results indeed all belong to the domain eliminates the need of human assistance for judging the results or comparing them with existing descriptions. A high precision is also a good basis for further improvements of the method. However, we also discuss some possibilities for improving the recall as well in the future work section.

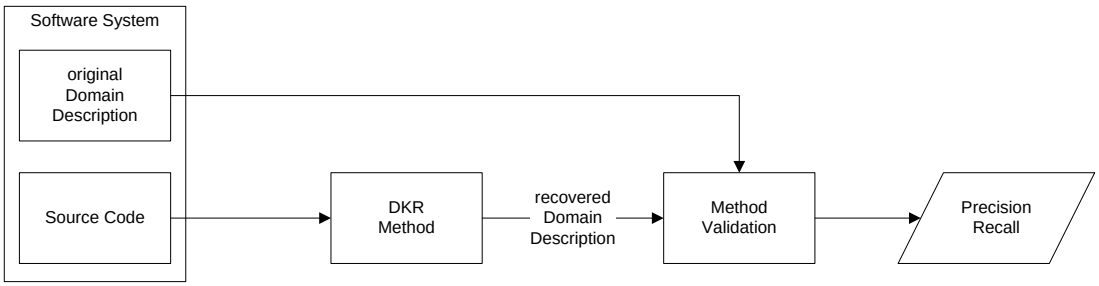


Figure 3.1: Framework for validation of Domain Knowledge Recovery methods

3.3 Formal Concept Analysis

Formal Concept Analysis (FCA) is a data analysis technique which is used for discovering latent concepts in sets of related data. These concepts are found by visually interpreting the resulting representation, which is a lattice. FCA has been described in detail in [12, 25]. The following section gives an overview of the definition.

3.3.1 Definition

FCA is based on a *formal context*, which is a triple $\mathbb{C} := (\mathcal{O}, \mathcal{A}, \mathcal{I})$ with a set of objects $\mathcal{O}$, a set of attributes $\mathcal{A}$, and the relation $\mathcal{I}$ on $\mathcal{O} \times \mathcal{A}$ so that $\mathcal{I} \subseteq \mathcal{O} \times \mathcal{A}$.

The set of common attributes for a set of objects $X \subseteq \mathcal{O}$ and the set of common objects for a set of attributes $Y \subseteq \mathcal{A}$ can be determined by using the *derivation operator* "^{I}". This derivation operator is applicable for both objects and attributes and it maps either a set of object to a set of attributes: $X \subseteq \mathcal{O} \mapsto Y \subseteq \mathcal{A}$; or a set of attributes to a set of objects: $Y \subseteq \mathcal{A} \mapsto X \subseteq \mathcal{O}$. It is defined as follows:

$$X^I := \{a \in \mathcal{A} \mid \forall o \in X : (o, a) \in \mathcal{I}\},$$
$$Y^I := \{o \in \mathcal{O} \mid \forall a \in Y : (o, a) \in \mathcal{I}\}.$$

For two sets Z_1 and Z_2, which are either both subsets of $\mathcal{O}$ or both subsets of $\mathcal{A}$, the derivation operator satisfies the following conditions:

$$(1) Z_1 \subseteq Z_2 \Longrightarrow Z_1^I \supseteq Z_2^I, (2) Z \subseteq Z^{II}, (3) Z^{III} = Z^I.$$

A *formal concept* is defined as a pair (O, A) with $O \subseteq \mathcal{O}$, $A \subseteq \mathcal{A}$, $A = O^I$ and $O = A^I$. O is called the *extent* and A is called the *intent* of the concept (O, A). A concept is therefore identified by its extent and its intent: the extent is the the set of all objects which belong to the concept and the intent is the set of attributes which is shared by all these objects.

The set of all formal concepts of a given formal context form a partial order, defined as $(O_1, A_1) \leq (O_2, A_2) \iff O_1 \subseteq O_2$, which is equivalent to $(O_1, A_1) \leq (O_2, A_2) \iff A_1 \supseteq A_2$, due to above stated conditions. This relation between concepts is also called the *subconcept-superconcept-relation*.

The *object concept* of an object $o \in \mathcal{O}$ is the concept (o^{II}, o^I), denoted with $\gamma(o)$, and is the smallest concept with o in its extent. The *attribute concept* of an attribute $a \in \mathcal{A}$ is the concept (a^{II}, a^I), denoted with $\mu(a)$, and is the largest concept with a in its intent.

The formal concepts of a formal context constitute a complete lattice, the *concept lattice* $\mathcal{L}(\mathcal{C})$. Every complete lattice has a top element and a bottom element. Identical formal contexts will result in identical concept lattices, thus: $\mathcal{C}_1 = \mathcal{C}_2 \iff \mathcal{L}(\mathcal{C}_1) = \mathcal{L}(\mathcal{C}_2)$.

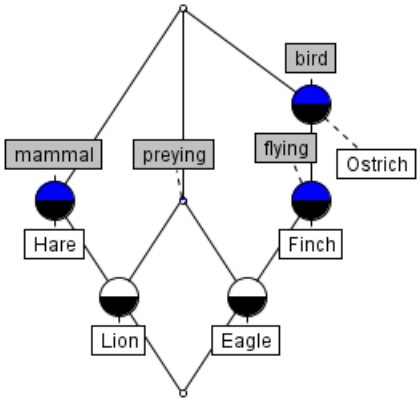
3.3.2 Input and Parameters

Input to FCA is a formal context, which is often depicted by a cross table, as shown in Figure 3.2. In most applications of FCA, the set of objects $\mathcal{O}$ (the rows in the table) and the set of attributes $\mathcal{A}$ (the columns in the table) are disjoint and their relation is defined as "object o has attribute a" (marked with an 'X' in the table if true). The resulting relations then represent a generalization or specialization between the related concepts. This has also been applied successfully for different software engineering activities, as described in [47].

However, FCA is not limited to the "has a"-relation. In principle, all sets of objects having related attributes can be used as input. This allows for variations in the usage of FCA, especially with respect to the analysis of source code. An example is described by Cole and Tilley in [15], which also used object-relation-attribute

A	B	C	D	E
	preying	flying	bird	mammal
Lion	X			X
Finch		X	X	
Eagle	X	X	X	
Hare				X
Ostrich			X	

(a) Cross table



(b) Hasse diagram

Figure 3.2: A formal context (shown as cross table) and its corresponding concept lattice (shown as Hasse diagram), created with ConExp [1].

combinations like method-defined in-class, package-calls-package, etc. The required information for constructing the context can be gathered by static source code analysis.

Using other relations as well as other object/attribute-combinations for constructing the context offers more possibilities for the interpretation of a resulting lattice. This way properties which are specific for an application area can be taken into account, and the lattice representation can offer new ways for exploring relationships and drawing conclusions.

3.3.3 Interpretation of the Results

A lattice shows the concepts, but also reveals some structures and relations which are latent in the context. It therefore can offer new insights into the information represented by the context. If e.g. a concept is located close to the top of the lattice, then this concept is a more general one, having a small amount of attributes which count for many objects (aka large extent and small intent). The concepts close to the bottom are more specific concepts with a large amount of attributes and not many objects (small extent and large intent).

Lattices are usually visualized using a *Hasse diagram* or *labelled line diagram* (see Figure 3.2 for an example) and interpreted by a human. While the content of the lattice stays the same, the layout of the visualization can differ. As lattices are mostly interpreted by observing the structure of the representation, the way the lattice is presented is therefore important.

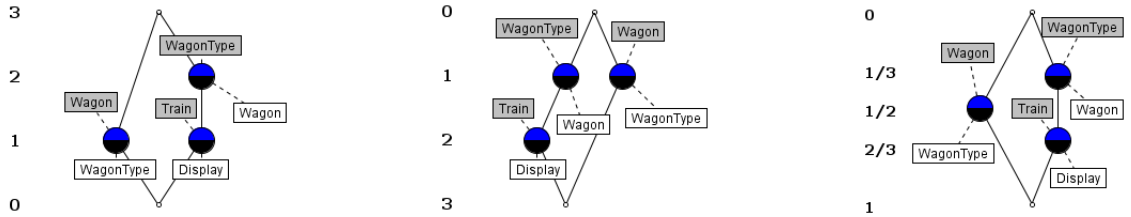
Lattice diagrams are vertically[2] sorted using the *Hasse diagram constraint* [14]: if an element a is less than an element b (which can be determined using the order relation of the concepts described above), then a occurs further down in the diagram than b. These relations form *chains* between the top and bottom of the lattice. So the position of a concept in a hierarchical order can be determined by the position of the concept in the chain with the largest possible number of elements. For the visualization of lattices, *ranks* are used for determining the concepts' position. These ranks can be automatically calculated, and therefore we will use ranks also for this work.

Different strategies are known for the calculations of the ranks. Cole et al. define these three in [14]:

- Uprank - The length of the longest path from an element to the top of the lattice.
- Downrank - The length of the longest path from an element to the bottom of the lattice.
- Average Rank - The concept position relative to the chain of the longest paths to the top and the bottom (not used in this work).

Depending on the strategy we get different lattice representations, which is shown in Figure 3.3. If e.g. the vertical position is relevant for the interpretation, then in Figure 3.3(a) the concepts with the objects "WagonType" and "Display" would have the same rank and in Figure 3.3(b) the concepts with the objects "Wagon" and "WagonType" would also have the same rank. While both are representations of the same lattice, their visualization strategy leads to different interpretation results. This shows that a grounded choice of a ranking strategy for all automatic interpretations of lattices is necessary.

[2]There are also algorithms for horizontal sorting, these are not used in this work and will not be discussed.



(a) Downrank (b) Uprank (c) Average Rank

Figure 3.3: Possible hierarchy assignments using different ranking strategies

The algorithm for determining the upranks of the elements of a lattice which is used for this work is given by Cole et al. [14] and shown in Equation 3.1. The calculation of downrank is similar to this and therefore not shown here.

$$\begin{aligned} uprank(\top) &= 0 \\ uprank(p) &= 1 + \max_{q \in L | q \prec p} uprank(q) \end{aligned} \tag{3.1}$$

To increase the comprehensibility, every object o and every attribute a of a formal context is only displayed once in the lattice representation and is connected to its object concept $\gamma(o)$ and attribute concept $\mu(a)$, respectively. The complete intent of a concept can be determined by collecting all attributes while traversing upwards in the lattice diagram. Similarly, the complete extent of a concept can be determined by collecting all objects while traversing downwards in the lattice diagram.

3.3.4 Applicability for (Automated) Domain Knowledge Recovery

FCA has been used successfully for different software engineering activities, a summary of these can be found in [47]. 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 [10, 17, 19, 44, 48]. However, the identified modules also include technical modules, so the described approaches have to be extended.

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 [11]. If applied for domain knowledge recovery, these properties can also be helpful for finding possible domain candidates.

All efforts reported in the FCA literature still require human assistance for the interpretation of the lattices, and often also human improvement of the diagrams is required in order to be satisfactory [14, 25]. As we have shown is the interpretation of a lattice dependent on its graphical representation. This has to be taken into account when the lattice diagram has to be interpreted automatically.

To our best knowledge, no efforts were made yet to automate this step for getting the results of the interpretation. Of course, in general one can not automate the step, but by specializing to the domain of domain recovery we might be able to add automation by explicitly making use of the properties of concepts in lattice diagrams and the visualization strategies.

One way this could be realized is by implementing the ranking strategies and functionality to extract the objects and attributes of concepts. The calculated ranks can then be used for assigning weights to the relevant concept parts (either objects or attributes). Such a domain knowledge recovery method would therefore combine the properties of domain knowledge and the lattice properties described earlier.

So, to obtain fully automated domain recovery we need to extend FCA by post-processing the lattice with additional automated analysis steps as described above. These analysis steps are the main contribution of this thesis.

3.4 Latent Dirichlet Allocation

Latent Dirichlet Allocation (LDA) is a generative probabilistic model for collections of discrete data such as text corpora and was first introduced by Blei et al. [7]. It is used for finding latent topics in a collection of text

documents. The following section gives an overview of the definition of LDA.

3.4.1 Definition

The basic parts of LDA are:

- *Words* - A word is the basic unit of data and part of a vocabulary.
- *Documents* - A document is a sequence of words.
- *Corpus* - A corpus is a collection of documents.

The documents in the corpus form the basis for the analysis. The words are extracted from the documents and then used for the LDA algorithm to identify the latent *topics*. A topic is a collection of words along with the importance of each word to the topic, represented as distribution by defining an occurrence probability [38]. Put another way, the words in a topic are likely to be found together (and distributed as defined by the occurence probabilities) in some documents in the corpus. So a topic is a common word distribution in a corpus of documents.

Words can occur in more than one topic, and also combinations of words can occur more often in different topics having different distributions. Because a topic is defined as combination of *all* its words *including* the probabilities, the problems of *synonymy* (multiple words with the same meaning) and *polysemy* (words with multiple meanings) are adequately addressed as described by nearly all authors.

The topics are related to the analysed documents, again using a probability distribution (equivalent to the distribution of words in a topic). In other words, documents are described by topic distributions. This way documents can be identified or categorized by looking for specific topics, which does e.g. say something about the similarity of two documents. Possible applications are for tasks as classification, novelty detection, summary, and similarity and relevance judgements [7].

Based on a corpus of documents – which consist of words – LDA therefore (1) identifies a set of topics, (2) associates a set and distribution of words with a topic, and (3) defines a mix of topics specific for each document in the corpus [38].

LDA is a non-deterministic method. There exist different algorithms for calculating the topic model and distribution, where variational Bayes, (collapsed) Gibbs Sampling, and expectation propagation are the most common ones [45]. All algorithms use approximation or convergence in an iterative process, so the results are depending on the amount of iterations and the starting sample (the first randomly chosen set of word-topic-assignments) and therefore non-deterministic.

3.4.2 Input and Parameters

In this section we will focus the discussion of input and parameters on the application of LDA specific for software engineering purposes.

The granularity of documents differs greatly depending on the goal of the LDA application. The whole source code of a system as one document was used for software categorization [46]. Source code files as documents were used for mining business topics and analyzing software evolution [38, 33]. Methods as documents were input for bug localization approaches and estimating the optimal number of topics [36, 37, 26]. Lukins et al. also describe the possibility of using packages or classes as documents. There are also combinations possible, e.g. the tool Topic$_{XP}$ offers the user to freely choose between using methods, classes, or packages as documents [43].

Basically, LDA is used for normal text files, so (nearly) all words are taken into account. However, source files are also text files, but with a defined structure and identifiable parts, as e.g. classifiers/identifiers, programming language keywords, parameters, comments, error messages, string literals, etc. Most authors did not take this into account for the extraction of the words. However, nearly all removed the keywords as part of a preprocessing (see list below). Only two papers explicitly took comments and error messages into account [36, 37].

These preprocessing steps are common for most described approaches:

- Splitting words - Identifiers often consist of more than one word, using camel case notation or separators for concatenating them. These words are split into their word components [43, 38]. This technique is also known as tokenisation, efforts for its improvement are described by Butler et al. in [9].
- Removing stop words - these are keywords of the programming language (“if”, “then”, “class” etc.), but also words that are commonly used like “get”, “set” etc. [43, 38].

- Stemming words - removing the endings from words so that they are independent of their grammatical context [43, 38].
- Weighing words - Maskeri et al. additionally are weighing the extracted keywords based on their occurence in a programming language specific context. E.g. classes represent domain objects in object oriented systems, so the keywords extracted from their names are of more importance than e.g. the attributes of the class. The result will be that these are more prominent in the resulting topics.

The amount of topics to be extracted has to be defined in advance and given as parameter to the LDA algorithm. This amount has great impact on the results [38]. Too few topics lead to over-generalization and too many topics may fail to bring related sets of code together and leads to having as many topics as documents [26]. Suggestions vary from the square root of the document size to magic numbers like 200 or 300. Grant and Cordy describe an approach to estimate the optimal number of topics (latent concepts as they call it) using Vector Space Neighbours and a Package Structure Heuristic [26]. If used for finding modules, then few topics can be expected to lead to better results.

Another parameter determines the inclusion of words in topics. It can be defined either as probability threshold, meaning that only words are selected with a probability greater than the predefined threshold, or as total amount x of words per topic, so that only the x words with the highest probabilities are included in the topic.

Besides the amount of topics also other parameters are required for LDA: α, which controls the division of documents into topics, and β, which controls the division of topics into words. These are not further discussed here.

Depending on the used algorithm for the calculation of the topics, the number of iterations has also to be given as parameter, e.g. if Gibbs sampling is used.

Usually identifiers are just split in their parts, but using the powerset of these parts could also help in finding more latent concepts by concatenating the elements of the powersets' subsets and using these as equivalents to the original identifier. Sets naturally are unordered, but the order of the original identifier parts can be used. If e.g. applied to the identifier "CreateWagonType", the concept "WagonType" would also be found here, which would not be the case with pure splitting. If used with the classnames "CreateWagonCommand", "CreateTrainCommand", and "RemoveWagonCommand", the latent concepts "CreateCommand" and "WagonCommand" will also be identified.

3.4.3 Interpretation of the Results

The results of LDA, the topics and the topic-document-distributions are usually used for activities like document modeling, text classification, or collaborative filtering. These activities only *use* the topics and distributions, and do not *interpret* them directly.

LDA has also been used for some tasks which required an interpretation of the found topics:

- *Mining business topics* - Maskeri et al. [38] identify topics in source code and relate them to source code files. Labeling topics has to be done manually in this approach as well as filtering and elimination of keywords that do not indicate any business concept. The topics they extract include also infrastructure-level topics and cross cutting topics. They also used an iterative process: initally extracted topics are evaluated and based on the results the parameters, keyword filtering are updated. However, the automated evaluation of the quality of the obtained topics is difficult. So their solution still requires much human assistance.
- *Mining Concepts* - Linstead et al. tried to find concepts (represented as topics) in source code [34]. Some of the found topics in their results were clearly summary topics (including technical topics), while others contained a lot of noise. The collection of these topics was completely automated, but the interpretation required human assistance. They state that the noise can be partly controlled by optimizing the number of topics given as parameter.
- *Topic explorer* - The tool Topic$_{XP}$ presented by Savage et al. [43] is an Eclipse plug-in implementation of LDA for the examination of latent topics in java systems. The users first have to define LDA parameters, then the tool extracts topics from a given system and visualizes the topics and their relationships. The generated topics can also be queried. This tool also shows the documents (classes or packages) most associated with (selectable) topics.

All applications of LDA, where the topics are subject to interpretation, were thus reported to require a high amount of human assistance, as the resulting topics include a lot of noise. The gathered results are

furthermore highly dependent on the parameters used for the LDA algorithm. Many applications reported in the literature required multiple iterations with a necessary human assisted adjustment of the parameters before getting sufficient results.

Research has been done on automatically determining these parameters, e.g. in [26], but the results are not sufficient yet.

3.4.4 Applicability for (Automated) Domain Knowledge Recovery

As stated earlier, all applications of LDA that use the topics for interpretation require a high amount of human assistance and are also highly dependent on the parameters used. The topics found contain a lot of noise in the form of containing words which do not reflect a coherent concept.

We also implemented LDA in Rascal, using the Java implementation of Gibbs Sampling JGibbsLDA [2] as algorithm. The examples were the students implementations of the RichRail system, as described earlier in this work. Different elements of source code were used as artifacts, like e.g. source files or only the content of methods. We also used different parameters for the algorithm, like varying numbers of topics, varying numbers of words per topic, and also varying numbers of iterations.

However, none of the experiments led to sufficient or promising results. If LDA was applied two times with exactly the same input and parameters, the results were always different — which approved the non-deterministic character of LDA. This leads us to the conclusion that with the current knowledge LDA is not applicable for reliable automated domain knowledge recovery and we therefore do not use it in our approach.

3.5 Related Work

Some research has been done on Domain Knowledge Recovery, often under different names. Diaz et al. called this Automatic Domain Analysis (ADA) [18]. They used all available artifacts for the analysis *besides* source code. Their approach furthermore 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.). So their focus is mainly on the relations between concepts and artifacts.

Li et al. use the source code as input for their domain knowledge recovery approach [32], the other input is 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.

All described approaches require human assistance (often from domain experts), especially for determining if the found concepts indeed belong to the domain. They also all use already described domain knowledge, which makes them more domain knowledge-artifact-association finding methods.

To the best knowledge of the author, no approach has been done on fully automated recovery of domain knowledge, which does not require human assistance end results in a representation of the domain knowledge implicitly contained in the source code of a software system. Our work intends to fill that gap and to prove that this is possible.

Chapter 4

The Domain Knowledge Recovery Method - DoKRe

The main goal is to automate the process of domain knowledge recovery as much as possible. In this chapter we describe the steps of our **Do**main **K**nowledge **Re**covery method — DoKRe — and their implementation using FCA. We focus thereby on the extraction of conceptual classes first, as these are the most important parts of a domain description (see section 3.1.2 for the discussion of all parts of a domain description).

4.1 Method Overview

The DoKRe method is intended to be a proof of concept that the properties of domain knowledge in source code in combination with automation possibilities of the interpretation of FCA results (the lattices), both discussed in chapter 3, can be used for domain knowledge recovery.

The method consists of a collection of steps. Most of these steps follow a common framework, but are individually executed per *ranking scheme*. A ranking scheme is the concrete implementation of these steps specific for one of the different combinations of domain knowledge properties and lattice interpretation. The order of the ranking schemes is unimportant, as they are independent from each other. Beside these individual steps are also some general steps which are executed once after execution of the steps of all ranking schemes. An overview of the method is shown in Figure 4.1.

It is important to note that we put more emphasis on optimizing the *precision* of our method and less emphasis on the *recall*. We argue that it has a higher priority to have reliably identified domain concepts than to have a result which includes all domain concepts, but also a high rate of false positives which have to be identified with human assistance. However, we discuss in our conclusions suggestions for improving the recall of our method.

Performance was not taken into account during the implementation of our method, as we put emphasis on showing that our approach gives good results and not on efficiency.

In the next section we describe the common framework of the method, followed by a section that describes the specific aspects of all six ranking schemes in detail.

Example

We use one of the RichRail case studies as example to demonstrate the steps of the method in more detail. The examples use the first ranking scheme only, as the steps in the other ranking schemes are similar.

4.2 Method Framework

4.2.1 Input

As discussed earlier, we use only the source code for the recovery of domain knowledge. At this moment the method is limited to Java projects in Eclipse that can be analysed using the meta-programming language Rascal [5].

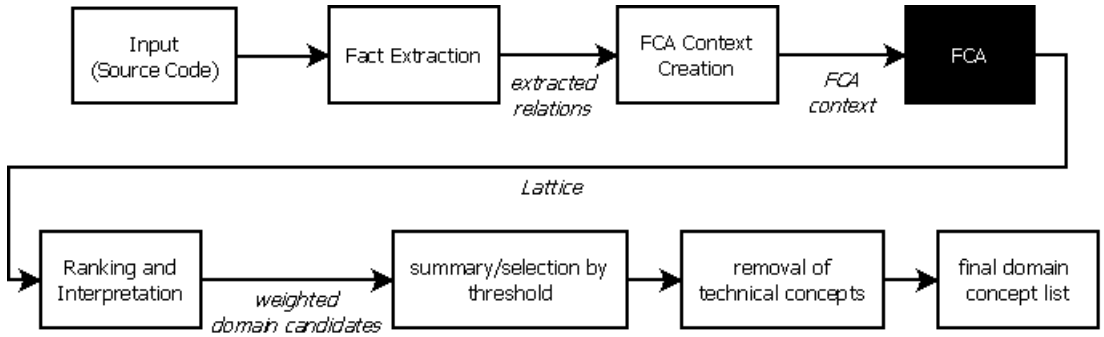


Figure 4.1: DoKRe Method - Overview

Example

The case system RichRail2 is imported as project in Eclipse. This project can then be used as input for the next steps.

4.2.2 Fact Extraction

The facts needed as input for the analysis are first extracted from the source code of the system. We use the functionality of Rascal for this, which first builds an internal representation of the analysed system as abstract syntax tree (AST). This AST can then be queried to get the necessary information, which are different source code elements like e.g., class names, package names, or method names. These elements are connected via binary relations like e.g., declares or calls. So the output of this step is always a set of source code elements pairs, whereby the first element is connected to the second element by a given binary relation.

Example

The names of all classes from the RichRail2 project and their declared fields, given as a set of (classname,fieldname)-pairs, form the result of this step[1]:

{ (Type, numberOfSeats),
(delete_command, command),
(List_display, txt1),
(RichRail, logdisplay),
(Wagon, id),
(Train, wagons),
(Station, types),
(Img_display, TRAINLENGTH),
(RichRail, command_button),
...
(Station, trains),
(Station, img_display),
(Station, wagons),
(Station, list_display),
(RichRail, imgdisplay),
(RichRail, listdisplay),
(Train, id) }

4.2.3 FCA Context Creation

The set from the fact extraction step is then used to create the input context for the application of FCA. In some cases this context is preprocessed in order to reflect the specific properties of domain knowledge in source code. In DoKRe, two different preprocessing variations are applied.

The first one is *tokenized names*, which splits the identifier name based on camelCase notation and common separators (like '_') as criterion. After splitting, all tokens are also capitalized and stemmed to the singular form using one of the steps of the Porter stemming algorithm [40]. An example of this is given in Figure 4.2. This example shows that after applying *tokenized names*, the domain concepts Wagon and Train now are attributes of more objects than the other attributes, which can be used in later steps of the method.

The second variation is *powerset of name*, which makes use of the tokenized names. It adds all permutations of the concatenated subsets of the tokenized name to the context — preserving the original order of the tokens

[1]Not the whole set is shown.

	AddWagonToTrain	drawWagon	drawTrain
Station	X		
Img_Display		X	X

(a) original context

	Add	Wagon	To	Train	Draw
Station	X	X	X	X	
Img_Display		X		X	X

(b) context using tokenized attribute names

Figure 4.2: An example of using tokenized attribute names in the context.

	WagonName	Train	TrainName	Wagon
CreateWagonSeat	X			X
RemoveWagon	X		X	

(a) original context

	WagonName	Train	TrainName	Wagon
CreateWagonSeat	X			X
RemoveWagon	X		X	
Create	X			X
Wagon	X		X	X
Seat	X			X
CreateWagon	X			X
WagonSeat	X			X
CreateSeat	X			X
Remove	X		X	

(b) context using powerset of object names

Figure 4.3: An example of adding the powerset of object names to the context.

— together with the original relations[2]. An example is given in Figure 4.3, 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. This property can then again be used in later steps of the method.

Example

In the example using the first ranking scheme, *powerset of name* is applied to both the objects and the attributes in the set. The created names are then added to the set by creating pairs containing the original element and the new name as the other element. One exemplary pair in the case study in the result of the fact extraction is (Type, numberOfSeats). After the application of powerset of name, the following pairs are added to the set:

{ (Type, OfSeat),
(Type, Number),
(Type, NumberOf),
(Type, Seat),
(Type, NumberOfSeat),
(Type, Of),
(Type, NumberSeat) }

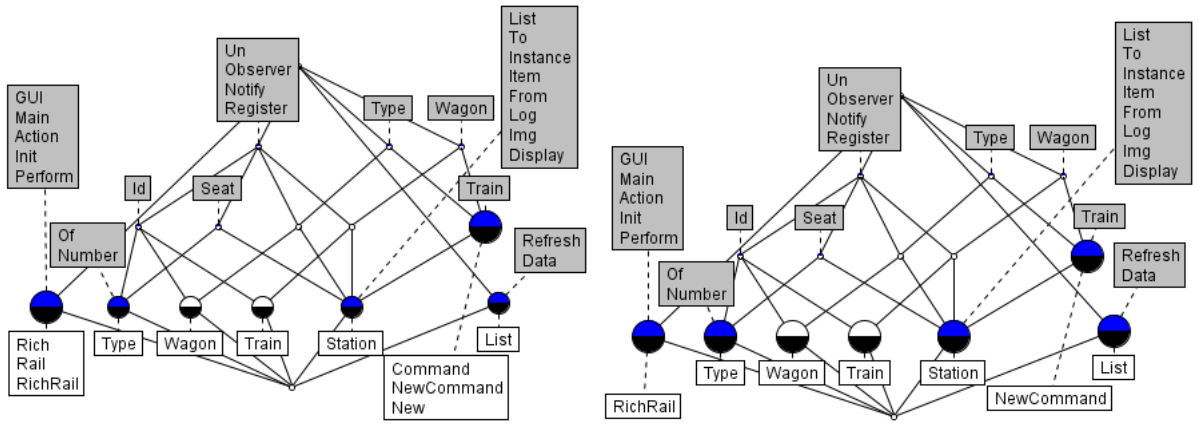
This enriched set of pairs is then transformed into a formal context and written to a file, using the common cxt file format from Burmeister [41].

4.2.4 FCA Application

As discussed in section 3.3.1, a lattice created from a specific context is unique for this context. In other words, the specific implementation of FCA does not have an impact on the result and is mostly interesting for performance improvements. As we do not consider performance in our method, we can treat the application of FCA as a *black box* which is using one of the possible implementations. In Figures 4.1 and 4.5 this is emphasized by using black boxes in the FCA application-step.

We used the implementation of FCA which already existed in Rascal. This implementation takes as input the context in the form of a cxt-textfile with a predefined format. The result is a lattice, which is internally represented as the set of all (concept,concept)-relations in the lattice.

[2]Note that different from set theory we preserve the order of the tokens and use the token permutations instead of combinations.



(a) lattice with powerset of names

(b) lattice with irrelevant powerset parts removed

Figure 4.4: An example of removing irrelevant parts of the powerset of names from the lattice.

Example

The application of FCA in the case study leads to a set of pairs of concepts, which are related by the superconcept-subconcept relation (see section 3.3.1). Each concept is defined by its extent and intent — the sets of common objects and attributes. Following is a part of the internal representation of the resulting lattice:

{ ...,
(({Imgdisplay,Station,Img,Display},{Wagon,Train}) , ({Imgdisplay,Train,Station,Img,Display},{Wagon})),
(({Wagon},{Id,Type,Observer}) , ({Wagon,Station},{Type,Observer})),
(({Wagon,Station},{Type,Observer}) , ({Wagon,Type,Train,Station},{Observer})),
(({Train},{Wagon,Id,Observer}) , ({Wagon,Type,Train},{Id,Observer})),
(({Train,Station},{Wagon,Observer}) , ({Imgdisplay,Train,Station,Img,Display},{Wagon})),
(({Wagon},{Id,Type,Observer}) , ({Wagon,Type,Train},{Id,Observer})),
(({Wagon,Type,Train},{Id,Observer}) , ({Wagon,Type,Train,Station},{Observer})),
(({Train},{Wagon,Id,Observer}) , ({Train,Station},{Wagon,Observer})),
(({Train,Station},{Wagon,Observer}) , ({Wagon,Type,Train,Station},{Observer})),
... }

4.2.5 Ranking and Interpretation

If the context is created using powerset of names, then a preprocessing step is executed prior to the actual ranking and interpretation of the lattice. Using powerset can result in larger amounts of objects. If the parts of the powerset are found in the same object concept in the resulting lattice, then they did not lead to the identification of relevant concepts through being part of identifier names (as they have the same attributes as the original identifier) and can therefore be removed. This decreases the amount of objects in the lattice without changing the relevant content. Figure 4.4 exemplary shows the removal of the objects “Rich”, “Rail”, “New”, and “Command” from the lattice.

At this moment, the DoKRe method only uses the *vertical* position of objects and/or attributes for the interpretation, especially the distance to top or bottom of the lattice. For the automatic determination of the vertical position we can use the earlier described ranking strategies.

After the optional preprocessing, the lattice is therefore ranked using either the downrank or uprank strategy (see section 3.3.3), depending on the specific aspects of the applied ranking scheme. A rank is thereby assigned to each concept in the lattice. These ranks are then used for the automatic interpretation of the lattice.

As described in section 3.3.3, a visual representation of a lattice shows all objects and attributes only once. The objects and attributes are thereby related to their object concept, respectively attribute concept (see section 3.3.1 for the definition of object concept and attribute concept). Using this relation, we can assign the earlier calculated ranks of concepts also to the corresponding objects and attributes. As every object has exactly one object concept, this leads to an explicit assignment of ranks to objects. The same applies for the attributes.

The objects or attributes with the assigned ranks are then translated into a list of domain candidates, whereby the ranks are transformed into weights. The highest possible weight is varying per ranking scheme and depends on the weight of this ranking scheme in the total weight for the method. The distribution of these weights, or the effectiveness of the specific ranking schemes in relation to the other ranking schemes, was defined by using the examinations of the results of test applications of the method.

Some ranking schemes use a threshold for the addition of domain candidates and their weights to the result, like e.g. only using the upper half of the lattice. Some ranking schemes apply a special filtering which reflects some special properties of the domain knowledge in source code, these are described in the respective sections.

The output of this step (per ranking scheme) is a list with domain candidates and the weights assigned to them.

Example

In our example we used the powerset of names, so we need the preprocessing step, which removes irrelevant attributes from the lattice. This is exemplarily shown in following example, where the objects which are marked in black are part of the name of another object and therefore removed.

before preprocessing:
(({Imgdisplay,Station,**Img**,**Display**},{Wagon,Train}) , ({Imgdisplay,Train,Station,**Img**,**Display**},{Wagon})),
after preprocessing:
(({Imgdisplay,Station},{Wagon,Train}) , ({Imgdisplay,Train,Station},{Wagon})),

After the preprocessing, the lattice in our example is ranked using the downrank strategy. The result is a mapping of each concept to a rank. Note that the preprocessing step is not applied to the intents — the attributes — of the concepts.

5:{({Imgdisplay, List, Logdisplay, Wagon, Type, Get, Getcommand, Station, RichRail, Command, Delete, ...},{})}
4:{({Imgdisplay, Train, Station, Img, Display}, {Wagon})}
3:{({Wagon, Type, Train, Station}, {Observer}),
({Imgdisplay, Station, Img, Display}, {Wagon, Train}),
({List, Logdisplay, Log, Listdisplay, Display}, {Txt})}
2:{({RichRail, Newcommand, Delete, Command, Factory, Addcommand, Removecommand, Get, Getcommand, ...),
({Wagon, Type, Train}, {Id, Observer}),
({Imgdisplay, Img, Display}, {CurrentTrain, TRAINLENGTH, Wagon, Current, Train, OFFSET, CurrentWagon}),
({RichRail, Station}, {Imgdisplay, Logdisplay, Listdisplay}),
({Train, Station}, {Wagon, Observer}),
({Logdisplay, Log, Display}, {Jsp, Txt}),
({Wagon, Station}, {Type, Observer})}
1:{({Station}, {Imgdisplay, Logdisplay, List, Wagon, Type, Train, Observer, Instance, Listdisplay, Log, Img, ...}),
({Type}, {NumberSeat, OfSeat, Id, Number, Of, Observer, NumberOfSeat, Seat, NumberOf}),
({Wagon}, {Id, Type, Observer}),
({RichRail}, {Imgdisplay, Textbox, Logdisplay, Text, Commandbutton, Mainpanel, Command, Box, Main, ...}),
({Display}, {CurrentTrain, TRAINLENGTH, Wagon, Current, Train, Jsp, OFFSET, CurrentWagon, Txt}),
({Train}, {Wagon, Id, Observer})}
0:{({}, {Imgdisplay, List, Id, Commandpanel, Of, Observer, Instance, Commandtextbox, CurrentTrain, ...})})

The assigned ranks are then transformed into weights. In the example, the objects — taken from the concept's extents — in the lowest rank 1 get assigned the highest weight 0.4, and only the two lowest ranks are used. This gives following end result for this case study and ranking scheme 1:

0.4: {RichRail, Wagon, Type, Train, Station, Display}
0.3: {Imgdisplay, Logdisplay, Command, Delete, Newcommand, Addcommand, Factory, Removecommand, Get, Getcommand, Remove, Log, Img, Add, Deletecommand, New}

4.2.6 Summary and Cleaning

The weights of the outputs per ranking scheme are summed up per domain class candidate, which gives a total list of domain candidates together with summed up weights from all ranking schemes. The higher the total weight, the higher is the probability that this candidate indeed belongs to the domain. This means that if the list is sorted descending by the weights, there is a certain point in the list where all candidates below this point can be excluded because of the unlikeliness that they indeed belong to the domain. This is realized by defining a *threshold* for the inclusion into the resulting list of domain candidates.

The observation of early tests of the method showed that in combination with the implemented ranking schemes a threshold of 1.5 leads to the best results. This value is therefore used in this work. A discussion about which value to use as threshold in general can be found in the conclusion chapter.

After the summary a *cleaning* of the result is performed. With cleaning we mean the removal of the obviously technical concepts from the result, which in some cases have properties in source code similar to these of domain knowledge and therefore occasionally also show up in the resulting list.

So this step removes the candidates which have a high probability of being technical or included only for design reasons. The basic implementation uses the Porter stemming algorithm [40] to stem the concepts in

order to also find spelling variations, and then excludes them from the result if they are part of following list of technical concepts: “log”, “logger”, “gui”, “exception”, “write”, “writer”, “read”, “reader”, “test”, “display”.

Also included in this list were keywords which are typically used for design pattern implementations [24]. These keywords are: “command” (Command pattern), “factory” (Factory pattern), “context” and “expression” (Interpreter pattern). This basic implementation can be improved, by e.g. using an approach for automatically detecting design patterns as described in [28, 39].

The final result is then a list of probable domain concepts together with weights.

Example

The following list is the complete log of the method application on RichRail2. Note that in this example no technical concepts had to be removed, as they did not get a weight higher than 1.5.

project:RichRail2
Facts from RichRail2 extracted
probs from ranking scheme 1:
0.4: {RichRail, Wagon, Type, Train, Station, Display}
0.3: {Imgdisplay, Logdisplay, Command, Delete, Newcommand, Addcommand, Factory, Removecommand, Get, Getcommand, Remove, Log, Img, Add, Deletecommand, New}
probs from ranking scheme 2:
0.6: {Wagon, Type, Station}
probs from ranking scheme 3:
0.5: {Wagon}
0.38: {Type, Train, Station}
0.25: {Log_display, Img_display, List_display}
probs from ranking scheme 4:
0.5: {Wagon, Type, Un, Notify, Observer, Register}
0.33: {Refresh, Id, Data, Train, Seat, Init}
probs from ranking scheme 5:
0.5: {Imgdisplay, RichRail, Wagon, Type, Factory, Train, Station, Img, Display}
0.33: {List, Logdisplay, Newcommand, Command, Listdisplay, Log, New}
probs from ranking scheme 6:
0.4: {RichRail, Wagon, Type, Train, Log_display, Observer, Img_display, Observable, List_display}
final probs:
2.9: {Wagon}
2.78: {Type}
2.01: {Train}
1.88: {Station}

The final list of domain concepts matches 100% with the reference model. The only difference is that depot is called station in this example.

4.2.7 Visualization

The visualization is not shown in Figure 4.1. The final list of domain class candidates is used to create UML diagrams, where the candidates are modelled as classes in a class diagram. We used the tool plantUML [4] for this. Examples are shown in chapter 5, where the results of the application of DoKRe are presented.

Example

The visualization of the recovered domain concepts of RichRail2 is shown in Figure 5.1.

4.3 Specific Ranking Schemes

The next sections describe all specific recovery steps of our approach in more detail. A detailed view of the method, which also reflects the specific steps, is given in Figure 4.5.

4.3.1 Ranking Scheme 1

This scheme makes use of the idea that the names of domain concepts can appear in more than one class name. The *fact extraction* gives the names of all classes and their declared fields (including the fields of their superclasses). For the *context creation* we then use the classnames as objects and the related fields as related attributes for the context. We also create the powersets of the classnames and add the relations of these with the original declared fields to the context (see section 4.2.3 for a detailed description and an example).

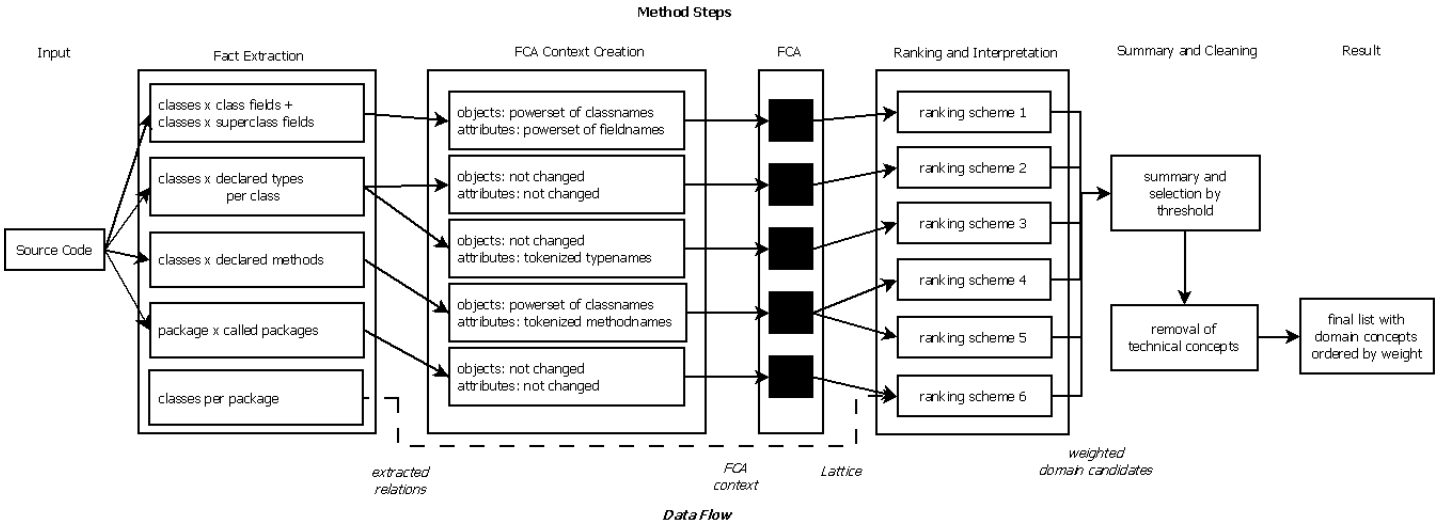


Figure 4.5: DoKRe Method - Detailed view

So, if a name of a domain concept is also part of other class names, then this will be also one of the added identifiers created by the powerset. The result is that the object in the lattice representing the domain concept now relates to the attributes of the other classes as well and will therefore have more related attributes in total. This usually makes the object move downwards in the lattice.

After the application of FCA, all objects are removed which were created by using the powersets of the classnames and which did not add relevant information. This is the case if these objects are still in the same context as the original object (with the original classname) and therefore have exactly the same attributes.

The lattice is then sorted using the downrank strategy. By observation of early tests we found that it is sufficient to use only the two lowest ranks of the lattice for the result. The weights are then assigned to the objects in these ranks, using 0.4 as maximum weight and 0.4 - (0.4/rankCount) as weight for the second rank. The objects and their assigned weights are the results of this scheme.

4.3.2 Ranking Scheme 2

In this scheme we use classes as objects and all declared types per class as attributes for building the context. Our hypothesis is that if classes are domain concepts then they probably will be used by many other classes (aka declared as type by many other classes), which will move them as attributes higher in the lattice. As this also counts for utilities and some general non-domain concepts, like e.g. Controller, Command, View, or Converter, these will also appear in the upper part of the lattice.

Our second hypothesis in this scheme is that domain classes will not make use of (all of the) technical classes, but technical classes will use some domain concepts.

So the fact extraction gives as classes and their declared types as relation, which are used unchanged as input for FCA. To implement the ideas behind our hypotheses, we take the concepts located directly under the top of the lattice (for this no sorting is needed). Then we use the combinations of these concepts and following algorithm for determining the domain candidates:

$$\begin{aligned} \forall (a_1, a_2) : a_1 \in C_1.intent, a_2 \in C_2.intent, C_1, C_2 \in (subconcepts\ from\ latticetop) \mid \\ a_1 \in C_2.extent, a_2 \notin C_1.extent \Longrightarrow a_2 \in (domaincandidates) \end{aligned} \tag{4.1}$$

This results therefore in a list of domain candidates, which are all given the weight of 0.6.

4.3.3 Ranking 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).

4.3.4 Ranking 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.

4.3.5 Ranking 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 o.5 as weight for the highest rank. All concepts are considered.

4.3.6 Ranking 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 4.5) 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.

4.4 Method Summary

The proposed method combines the possibilities which Formal Concept Analysis offers and the properties of domain knowledge in source code. The implementation of these different properties (or combinations of them) is realized with different ranking schemes, which use varying input, ranking, and interpretation.

The method is implemented in Rascal [5] and applied on some case studies. This implementation can be used to find out if (a) our assumptions about the properties of domain knowledge in source code are true and (b) that these properties can be used to recover domain concepts from source code. We furthermore want to show that it is possible to automate the interpretation of a visualized lattice, if the interpretation is based on the vertical position of the concepts.

For the evaluation of the method we use the two criteria: precision and recall. In order to be able to calculate both values for the case study systems, we need reference domain descriptions of these systems. The precision and recall can then be calculated using following formulas:

$$\begin{aligned} precision &= \frac{tp}{tp + fp} \\ recall &= \frac{tp}{tp + fn}. \end{aligned} \tag{4.2}$$

The *true positives* (tp) are the concepts which are recovered and also belong to the reference domain description. The *false positives* (fp)are the concepts which are recovered but do not belong to the reference domain description. The *false negatives* (fn) are the concepts of the reference domain description which are not recovered.

Chapter 5

Experiments and Results

This chapter first describes the experiment setup and the experiments which were executed using the various case studies. Then we present and discuss the results of the application of the DoKRe method.

5.1 Case Studies

The RichRail projects described in Section 2.3 are the initial experiments and used for the evaluation of the DoKRe method as proof of concept. These projects are suited as they differ broadly in the technical realization and can therefore be used to find out if the method can handle different ways of implementing the same domain. The domain of RichRail mainly consists of four concepts (see Figure 2.2), which allows an easy evaluation and validation of the method. The size of the RichRail projects comprises on average 17 classes and 4 packages.

For validation of the method in a more realistic setting, two medium sized systems from different domains are used: JHotDraw and SynPos. JHotDraw is a framework in the graphics domain and comprises 306 classes and 19 packages. SynPos is an open source implementation of a point-of-sale system and comprises 179 classes and 7 packages. The domains of both are well known and in the case of SynPos also partially described.

5.2 Experiment Setup

A prototypical implementation of the DoKRe method is done in Rascal [5]. Rascal is a meta-programming language, developed at the Centrum voor Wiskunde en Informatica (CWI), and offers the needed basic functionality for fact extraction from source code and the analysis of these facts. It uses the Eclipse development environment as host.

A basic FCA implementation already existed in Rascal, which is also used in this project. This implementation takes a .cxt-file as input and produces a lattice, internally represented as ⟨superconcept,subconcept⟩-pairs. The functionality needed for ranking and interpretation of the lattice was added to the Rascal FCA implementation.

For the experiments with LDA, the Java implementation JGibbsLDA [2] was mapped to Rascal.

The case studies were imported as projects in Eclipse and then analysed using the Rascal facilities. In the next sections we present the case studies and the results of the application of the DoKRe method.

5.3 Results from RichRail Projects

We analysed 22 RichRail projects which are described in Section 2.3. We did our experiments and adjustments of the method using the first 12 projects, and used the other 10 projects to objectively evaluate the DoKRe method. The results in terms of precision and recall are presented in Table 5.1. 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.

The average precision and recall over all RichRail projects are shown in Table 5.2. 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.

	precision in %	recall in %
RichRail1	100	75
RichRail2	100	100
RichRail3	100	100
RichRail4	100	75
RichRail5	100	75
RichRail6	100	100
RichRail7	100	100
RichRail8	100	75
RichRail9	100	100
RichRail10	67	50
RichRail11	100	50
RichRail12	100	50
RichRail13	100	100
RichRail14	100	75
RichRail15	100	75
RichRail16	100	50
RichRail17	100	75
RichRail18	100	75
RichRail19	100	75
RichRail20	100	100
RichRail21	100	100
RichRail22	29	50

Table 5.1: Results for recovery of conceptual classes from student projects

	avg. precision in %	avg. recall in %
RichRail1-12	97,25	79,17
RichRail13-22	92,90	77,50
total	95,27	78,41

Table 5.2: Average precision and recall of RichRail projects

Figure 5.1 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.

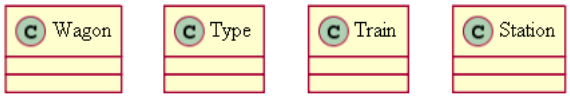


Figure 5.1: The recovered conceptual classes from RichRail2 in UML

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).

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 5.1, 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.

Figure 5.2: The recovered conceptual classes from RichRail10 in UML

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

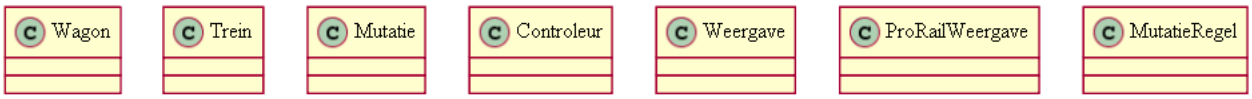


Figure 5.3: The recovered conceptual classes from RichRail22 in UML

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.

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.

Conclusion

The results show that the method gives a high precision for small systems, which we consider as important for an automated domain knowledge recovery method. The DoKRe method also is able to handle different technical implementations of a small domain.

5.4 Result from JHotDraw

Riehle documented in his dissertation some central abstractions of the JHotDraw framework [42], based on the JHotDraw tutorial and shown in Figure 5.4. As these can be seen as part of the domain model from JHotDraw, we can use them as reference for determining the effectiveness of our method. However, as there is no complete domain model from JHotDraw, our results will just give an indication of the effectiveness of applicability on larger systems.

The analysis was executed using JHotDraw version 6.

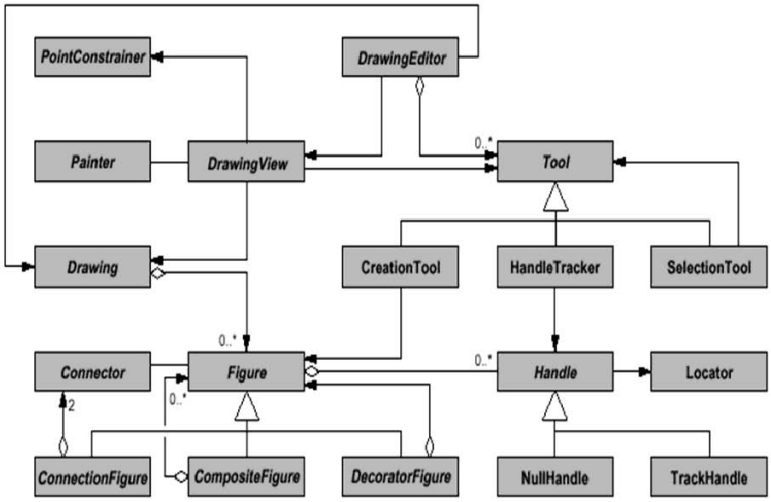


Figure 5.4: The central abstractions of JHotDraw, from [42]

Table 5.3 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.

Total Weight	**Conceptual Classes**
2.3	**Figure**
2.26	**Handle**
2.08	**Locator**
2.07	**Tool**
2.04	**Connector**
1.92	StorableInput, CollectionsFactory
1.88	*Geom*
1.81	*RelativeLocator*
1.80	*Attribute*
1.79	Event
1.77	**CreationTool**
1.75	*Draw*
1.74	Change
1.71	*Box*
1.69	*Connection*
1.62	*Text*
1.61	Helper
1.59	Iconkit
1.56	*View*
1.53	Selection
1.51	*Line*, *Color*
1.46	**CompositeFigure**, VersionManagement
1.45	Factory
1.42	Desktop
1.41	**Painter**

Table 5.3: Recovered conceptual classes from JHotDraw

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.

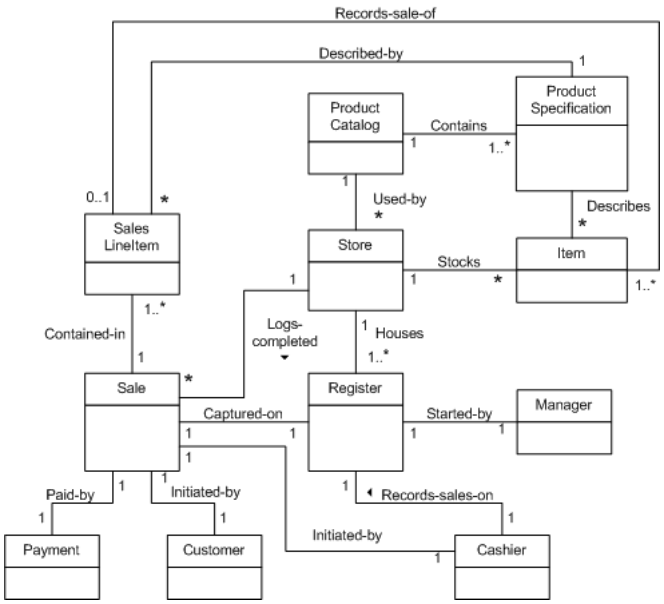


Figure 5.5: The domain model of the NextGen POS case study ©Craig Larman [31]

Total Weight	**Classes**
2.07	**User**
2.05	**Customer**
1.99	**Station**
1.9	SwingWorker
1.85	*CreditCard*
1.82	Synchronizer
1.81	**Product**
1.57	**Payment**
1.51	**Order**
1.5	StoreDB, I18N

Table 5.4: Recovered conceptual classes from SynPOS

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.

5.5 Result from SynPOS

Larman uses in [31] 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 5.5.

The case study in Larman's book was fictional, so we looked for an open source implementation of another POS system as this will likely use the same domain model. SynPOS [6] is an open source implementation in Java and was used as second larger system where we recovered the domain knowledge from.

The results of the application of our method on SynPOS are shown in Table 5.4 and Figure 5.6.

We assume that CreditCard is also part of the domain, but it is not included in the domain model as described by Larman [31]. This leaves four non-domain concepts: SwingWorker, Synchronizer, StoreDB, and I18N.

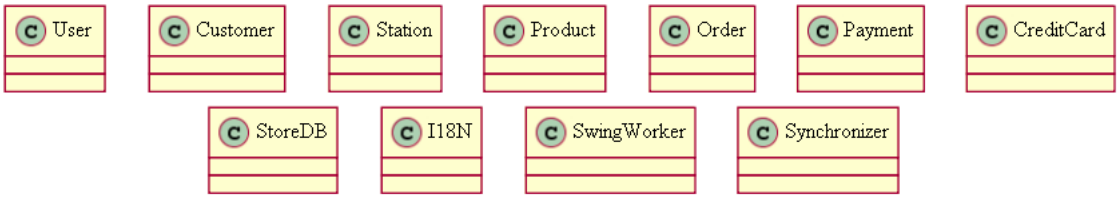


Figure 5.6: The recovered conceptual classes from SynPOS in UML

The precision for the analysis of the SynPOS system is therefore 64,6%. The recall will not be calculated, as Larmans' 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 cource 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.

Chapter 6

Discussion

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.

6.1 Evaluation of the method

6.1.1 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 5.3, 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.

6.1.2 Domain Knowledge Properties in Source Code

In general, the method just uses a small subset of possible applications of the intuitional 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.

6.1.3 Prerequisites and Limitations

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 powersets of their name parts) and the combination with lattices, the performance of the system is not sufficient for large systems at this moment.

6.2 Threats to Validity

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 [31]), 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.

6.3 Future Work

First, we only used 6 different ranking schemes. This present only a small subset of all possibilities. Our method shows that it is worth to explore and evaluate these other possibilities as well.

The automatic detection of technical concepts should be improved. By doing so, the results would become more reliable. There is also a chance that the threshold could be decreased, which could lead to a higher recall for the same precision.

Our methods recovers at this moment only conceptual classes. Future work should also look at the other parts of domain knowledge as defined in our domain metamodel, like e.g. the attributes of conceptual classes, associations, or hierarchy relations.

The method now uses static analysis only. By using dynamic analysis as well one could look at which recovered domain concepts really are used and if they are used in ways other than expected. This could lead to changes in behavior, attributes or associations of conceptual classes, as some of these parts are probably modelled, but never used.

Heuzeroth et al. described in [28] an approach for automatic design pattern detection. This can also be applied to the source code in order to improve the removal of technical concepts by identifying them as (parts of) design patterns. If patterns are detected, then assumptions can be made about which parts of their implementation possible domain candidates are, based on the roles played by the pattern participants.

6.4 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.

Bibliography

[1] ConExp Concept Explorer. `http://conexp.sourceforge.net/`, 2011.

[2] GibbsLDA++. `http://gibbslda.sourceforge.net/`, 2011.

[3] JHotDraw. `http://www.jhotdraw.org/`, 2011.

[4] PlantUML. `http://plantuml.sourceforge.net/`, 2011.

[5] Rascal. `http://www.rascal-mpl.org/`, 2011.

[6] SynPOS. `http://sourceforge.net/projects/synpos/`, 2011.

[7] Blei, D. M., Ng, A. Y., and Jordan, M. I. Latent dirichlet allocation. *J. Mach. Learn. Res. 3* (2003), 993–1022.

[8] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern-oriented Software Architecture - A System of Patterns*. John Wiley & Sons, Chichester, 1996.

[9] Butler, S., Wermelinger, M., Yu, Y., and Sharp, H. Improving the tokenisation of identifier names.

[10] Canfora, G., Cimitile, A., Lucia, A. D., and Lucca, G. A. D. Decomposing legacy systems into objects: an eclectic approach. *Information and Software Technology 43*, 6 (2001), 401–412.

[11] Caprile, B., and Tonella, P. Nomen est Omen: Analyzing the Language of Function Identifiers. In *Proc of the Working Conference on Reverse Engineering WCRE* (1999), IEEE Computer Society Press, pp. 112–122.

[12] Carpineto, C., and Romano, G. *Concept Data Analysis: Theory and Applications*. John Wiley and Sons, 2004.

[13] Chikofsky, E. J., and Cross II, J. H. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software 7*, 1 (1990), 13–17.

[14] Cole, R., Ducrou, J., and Eklund, P. Automated layout of small lattices using layer diagrams. *Formal Concept Analysis* (2006), 291–305.

[15] Cole, R., and Tilley, T. Conceptual analysis of software structure. In *Proceedings of Fifteenth International Conference on Software Engineering and Knowledge Engineering, SEKE03*, Citeseer, pp. 726–733.

[16] de Souza, S. C. B., Anquetil, N., and de Oliveira, K. M. A study of the documentation essential to software maintenance. *Proceedings of the 23rd annual international conference on Design of communication documenting & designing for pervasive information - SIGDOC '05* (2005), 68.

[17] Deursen, A. V., and Kuipers, T. Identifying objects using cluster and concept analysis. In *Proc of the Int Conf on Software Engineering* (1999), vol. 99, ACM, pp. 246–255.

[18] Diaz, I., Llorens, J., Genova, G., and Fuentes, J. M. Generating domain representations using a relationship model. *Information Systems 30*, 1 (Mar. 2005), 1–19.

[19] Eisenbarth, T., Koschke, R., and Simon, D. Locating Features in Source Code. *IEEE Transactions on Software Engineering 29*, 3 (2003), 210–224.

[20] Endres, A., and Rombach, D. *A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories*, vol. 21. Addison Wesley, 2003.

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

[22] Forward, A., and Lethbridge, T. C. The relevance of software documentation, tools and technologies. *Proceedings of the 2002 ACM symposium on Document engineering - DocEng '02* (2002), 26.

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

[24] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns*. Addison-Wesley, Boston, MA, 1995.

[25] Ganter, B., Stumme, G., and Wille, R., Eds. *Formal Concept Analysis, Foundations and Applications* (2005), vol. 3626 of *Lecture Notes in Computer Science*, Springer.

[26] Grant, S., and Cordy, J. R. Estimating the Optimal Number of Latent Concepts in Source Code Analysis. In *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on* (2010), pp. 65–74.

[27] Hay, D., and Healy, K. A. Defining Business Rules ~ What Are They Really ? the Business Rules Group. *Business* (2000), 4–5.

[28] Heuzeroth, D., Holl, T., Hogstrom, G., and Lowe, W. Automatic Design Pattern Detection. *MHS2003 Proceedings of 2003 International Symposium on Micromechatronics and Human Science IEEE Cat No03TH8717* (2003), 94–103.

[29] Jurkevičius, D., and Vasilecas, O. Ontology creation using formal concepts approach. In *Proceedings of the 11th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing on International Conference on Computer Systems and Technologies* (New York, NY, USA, 2010), CompSysTech '10, ACM, pp. 64–70.

[30] Köppe, C. Observations on the Observer Pattern. In *Proceedings of the 17th Conference on Pattern Languages of Programs* (New York, NY, USA, 2010), PLoP '10, ACM.

[31] Larman, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[32] Li, Y., Yang, H., and Chu, W. A concept-oriented belief revision approach to domain knowledge recovery from source code. *Journal of Software Maintenance and Evolution: Research and Practice 13*, 1 (Jan. 2001), 31–52.

[33] Linstead, E., Lopes, C., and Baldi, P. An Application of Latent Dirichlet Allocation to Analyzing Software Evolution. *2008 Seventh International Conference on Machine Learning and Applications* (2008), 813–818.

[34] Linstead, E., Rigor, P., Bajracharya, S., Lopes, C., and Baldi, P. Mining Concepts from Code with Probabilistic Topic Models Categories and Subject Descriptors. *Analysis* (2007), 461–464.

[35] Llorens, J., Morato, J., and Génova, G. *RSHP: An information representation model based on relationships*. Springer, 2004, pp. 221–253.

[36] Lukins, S. K., Kraft, N. A., and Etzkorn, L. H. Source Code Retrieval for Bug Localization Using Latent Dirichlet Allocation. In *2008 15th Working Conference on Reverse Engineering* (2008), IEEE, pp. 155–164.

[37] Lukins, S. K., Kraft, N. A., and Etzkorn, L. H. Bug localization using latent Dirichlet allocation. *Inf Softw Technol 52*, 9 (2010), 972–990.

[38] Maskeri, G., Sarkar, S., and Heafield, K. Mining business topics in source code using latent dirichlet allocation. In *Proceedings of the 1st India software engineering conference* (New York, NY, USA, 2008), ISEC '08, ACM, pp. 113–120.

[39] Pettersson, N., Lowe, W., and Nivre, J. Evaluation of Accuracy in Design Pattern Occurrence Detection. *IEEE Transactions on Software Engineering 36*, 4 (July 2010), 575–590.

[40] Porter, M. F. An algorithm for suffix stripping. *Program 14*, 3 (1980), 130–137.

[41] Priss, U. FcaStone - FCA file format conversion and interoperability software. In *Conceptual Structures Tool Interoperability Workshop (CS-TIW)* (2008).

[42] Riehle, D. *Framework Design: A Role Modeling Approach*. PhD thesis, ETH Zürich, 2000.

[43] Savage, T., Dit, B., Gethers, M., and Poshyvanyk, D. TopicXP: Exploring topics in source code using Latent Dirichlet Allocation. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance* (Washington, DC, USA, 2010), ICSM '10, IEEE Computer Society, pp. 1–6.

[44] Siff, M., and Reps, T. Identifying Modules via Concept Analysis. *Transactions on Software Engineering 25*, 6 (1999), 749–768.

[45] Teh, Y., Newman, D., and Welling, M. A collapsed variational bayesian inference algorithm for latent dirichlet allocation. *Advances in neural information processing systems 19* (2007), 1353.

[46] Tian, K., Revelle, M., and Poshyvanyk, D. Using Latent Dirichlet Allocation for automatic categorization of software. *2009 6th IEEE International Working Conference on Mining Software Repositories* (2009), 163–166.

[47] Tilley, T., Cole, R., Becker, P., and Eklund, P. A Survey of Formal Concept Analysis Support for Software Engineering Activities. In *In Gerd Stumme, editor, Proceedings of the First International Conference on Formal Concept Analysis - ICFCA'03* (2003), Springer-Verlag.

[48] Tonella, P., and Potrich, A. *Reverse Engineering of Object Oriented Code (Monographs in Computer Science)*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 2004.

Appendix A

Domain Models in the Literature

Evans described a couple of domain concepts which can be used for domain driven design [21]. Larman pays in [31] much attention on domain modelling. Fowler included two patterns on domain modelling in [23]: Domain Model and Transaction Script. Jurkevičius and Vasilecas summarize the elements of an ontology for describing domains [29]. Diaz et al. used the RSHP model presented in [35] for representing the domain of a software system [18]. Li et al. present in [32] a semantic network as basis for their domain knowledge recovery approach. The following section discusses the commonalities and differences of these domain descriptions. This discussion forms the basis for the definition of a domain metamodel (this is described in the context chapter), a summary is also given in Table A.1.

Generic Parts of a Domain

All authors describe the idea of *Conceptual Classes*, even though using different names for them. Evans distinguishes between Entities and Value Objects, where the main difference is that Entities have an identity and Value Objects only encapsulate state. The reasons for this distinction are mainly because of implementational aspects — Evans tightly connects the domain concepts with design concepts in his domain driven design approach — and therefore this distinction is not necessary for the definition of pure domain models.
Larman directly uses the term Conceptual Classes, while Fowler uses Objects. As objects are concrete instances of classes — if looking at object-orientation — and we only are interested in the generic concepts of the domain, we are not using the term objects to describe these concepts. Jurkevičius and Vasilecas distinguish between classes and individuals, but as in their approach individuals are equivalent to objects or instances of classes, this term will not be used due to the same reasons mentioned earlier. Diaz et al. talk generally about Information Elements (as part of the RSHP model). These Information Elements can be of many types, including Conceptual Classes.

The Conceptual Classes have *Attributes*, which represent the properties a conceptual class can have in order to represent a certain state. Some authors only implicitly mention attributes. Others also describe important properties of attributes: they are mostly numbers or text and they do not represent associations. Especially the latter one is important, as including attributes which represent an association and the association itself (see below) in the domain model would lead to redundant and/or ambiguous information and this should be avoided. In the RSHP model, attributes are also information elements.

Most authors also see the *Behavior* of conceptual classes as part of the domain, where behavior is named explicitly, as done by Evans, or as the kinds of events a class can react on, as done by Jurkevičius and Vasilecas. This behavior includes the responsibilities of a conceptual class, which is modelled in the form of methods. Fowler makes the behavior more concrete by explicitly adding calculation logic to it.
Larman excludes behavior from the domain model. He states that responsibilities and methods are not suitable in a domain model, as responsibilities usually are related to software objects and methods are purely a software concept, which becomes important only during design time. In order to keep the model more flexible, we also include the behavior of conceptual classes in it, as long as this behavior really is part of the domain and not of the technical realisation. In the RSHP model, (different types of) behavior are information elements too.

The conceptual classes are connected via *Associations*. Evans looks at associations already from an implementational viewpoint and suggests to constrain them as much as possible. According to him, most associations in real life are actually many-to-many and bi-directional associations, but should be . Larman stresses that associations are a "statement that a relationship is meaningful in a purely conceptual perspective–in the real domain" [31]. He defines the same constraints and properties for associations as also specified in the UML, namely direction, multiplicities, roles, and an association name.

The RSHP model has relationships between the information elements as the main concept. These relationships can be equivalent to associations, but also model the connection between conceptual classes and their attributes and behavior – as these are information elements too in the RSHP model. The properties of the association are in this model implicit in its type, and these types are again the ones defined in the UML as well (e.g. aggregation, composition, etc.), but also add a few other types like equivalence, linguistic, or qualification relationships.

Fowler and Larman distinguish between associations and *Hierarchy Relations*. In the RSHP model, the hierarchy relation is one of the relationship types and in the ontology of Jurkevičius and Vasilecas it is one of the ways in which classes can be related to each other. Evans does not mention hierarchy relations. But as they are fundamentally different from the earlier described associations, they are included as an apart component in the domain metamodel.

Services are only described by Evans. They are functionality which can not be connected to a specific conceptual class and are therefore modelled independent. Services are stateless and declare interfaces. Fowler describes the pattern Transaction Script, which is in essence the same as the services from Evans. These services can contain important knowledge of the business domain, and are therefore also included in the metamodel.

Evans introduces *Modules*, which are described by Larman as logical architecture elements – layers, subsystems, packages, etc. The main reason to use these is to group cohesive responsibilities and a separation of concerns. These also can be found in the business in the form of e.g. departments, sections, etc. Their names should reflect insight into the domain. They can be especially helpful in larger domains, as they help to keep the overview. Modules can also be defined in the RSHP model. The other authors do not explicitly mention them.

Constraints, often called Business Rules, play an important role in a business domain. The ontology of Jurkevičius and Vasilecas distinguishes between rules, axioms, and restrictions. These cover different aspects of the domain elements, as they can be applied to conceptual classes, their attributes and behavior, but also associations, services, and the other metamodel elements. Larman only mentions attribute requirements, while Evans only describes constraints on associations. Fowler uses the term validation logic for describing some parts of the constraints. An extensive definition and discussion of business rules can be found in [27].

Table A.1 gives a summary of the discussion and shows the relations between the elements of all different domain definitions and their mapping to the metamodel elements. The domain metamodel is described in the context chapter of this thesis.

Importance of Model Parts

Using the above described summary we also defined the importance of each model part. If a part is common to all domain descriptions found in the literature, then we considered that as very important. The fewer it was mentioned, the less important it becomes in general. This importance is thus purely based on the mentions in the literature, and can be different for specific domains. This importance list can also be used to determine where to start with when describing a new domain or when recovering an existing domain.

Conceptual classes are mentioned by all authors. Nearly all other parts of the model are not applicable without corresponding conceptual classes, we therefore consider them as the most important part of a domain model.

The most basic domain model will describe conceptual classes only, e.g. in the form of a glossary. Therefore are conceptual classes the only obligatory part of a domain model. All other parts are optional and can be filled in depending on the purpose of the model and the accuracy of the domain description needed in a specific situation.

The next important parts are the attributes of, and associations between, conceptual classes. These also can be found in all descriptions in the literature. Inheritance relations are mentioned explicitly only in a subset of the published models. But in the other models they are implicitly described as special kind of associations. We therefore consider them as less important than normal associations.

All models describe constraints, but differ in the interpretation of what these constraints actually contain. As we treat constraints in a quite general way, we consider the constraints related to attributes and associations as more important than constraints which represent business rules or general restrictions of other domain parts.

The behavioral aspects of conceptual classes are considered more important than general behavior which is modelled in the form of services. Reason for that, as stated by e.g. [21] that one first starts to describe the behavioral aspects and then determines which conceptual class is responsible for it. If no conceptual class seems to have this responsibility, it probably becomes a service.

Then the more general business rules and restrictions should be described. These usually have impact on or make use of the other parts of the domain, and should therefore be described at a later point.

Even if not mentioned by all authors, modules could help in getting an overview of the domain. A module is a cohesive set of domain concepts on a higher abstraction level. However, they often are more artificial concepts

Evans[21]	**Fowler**[23]	**Larman**[31]	**Jurkevicius**[29]	**Diaz et al.**[18]	**Domain Metamodel**
Entities (E)	Objects (O)	**C**onceptual **C**lasses	Classes	Information Elements (IE): nouns	**C**onceptual **C**lasses
Value Objects (VO)			Individuals		
E+VO: Attributes	O: Attributes	CC: Attributes	Attributes	IE: nouns	CC: Attributes
E+VO: Behavior	O: Calculation logic		Events	IE: nouns	CC: Behavior
Associations	Associations	Associations	Relations	IE: verbs + annotations	Associations
	Inheritance relation			IE: verbs + annotations	Inheritance relations
Services	Transaction Scripts			IE: nouns	Services
Modules				IE: nouns	Modules
Constraints on Associations	O: Validation logic	attribute requirements	Restrictions	IE: nouns	Constraints
			Rules		
			Axioms		

Table A.1: Summary of domain definitions and mapping to metamodel

than that they really match with some real abstractions in the domain and therefore often do not add something to the domain. So we consider them as the least important parts of the model.