

Methods to Create and Use Cross-Domain Analysis Patterns

Alexander Fülleborn and Maritta Heisel

Universität Duisburg-Essen, Fakultät für Ingenieurwissenschaften

Bismarckstrasse 81, D-47048 Duisburg

Phone +49 (0203) 379 3465

Fax +49 (0203) 379 4490

Email: alexanderfuelleborn@hotmail.de

maritta.heisel@uni-duisburg-essen.de

Abstract

We present a set of methods to enable a cross-domain reuse of problem solutions via *analysis patterns*. First, *problem-context-descriptions* and *problem-context-models* as well as *solution models* are used to express the domain-specific problems and their assigned solutions. After that, the *two-step abstraction method* is used to create cross-domain analysis patterns for the problem-context-models as well as for the solution models. The problem-context-patterns are used to search across domains for a solution pattern. If a solution pattern is available it can be instantiated in the solution seeking domain.

1 Introduction

To some extent analysis patterns are very different to those you can find for the architectural, design or implementation phases of the software engineering process: They reflect the real world with its numerous specific domains like business, technology, nature and so forth. These domains have their own terms and even their own languages, which also describe their proprietary problems and solutions for these problems. But what about *common* inter-disciplinary problems and solutions? There are already good examples for learning from other disciplines: technicians can adapt constructive principles from biology, which is described by *Bionics*. Other examples are given by methods such as TRIZ (Theory of Inventive Problem Solving) [TZZ00] that are used in construction.

Analysis models are expected to mirror the specific domains that have been mentioned before. Furthermore, there are problems and appropriate solutions that are valid also for other domains. Hence, such analysis models can help to find cross-domain solutions. Thinking in these terms, the analysis phase of software development can be viewed as an extended workbench for the domain specialist. Reusable analysis models (=analysis patterns) reflect the knowledge base of this extended workbench. Certainly also other authors like Fowler [Fow98] implicitly consider this cross-domain aspect. But with our work we describe a systematic process of creating cross-domain analysis patterns. To some extent, this systematic abstraction process can also be automated via tool-support.

To illustrate the proposed methods we use a case study coming from business domains. The principle is illustrated by Figure 1. Domain *A* is the solution seeking domain. Within the domain itself no solution exists for the requirement *A1*. In domain *B* a solution exists for the requirement *B2* that is also suitable to fulfill the requirement *A1*. The challenge is to make this solution available for *A1* in spite of the different vocabulary.

The rest of the paper is organized as follows: In section 2, we introduce our case study consisting of two requirements documents from different business domains. The methods to analyze and model the domain-specific problems and to enable a cross-domain reuse by an abstraction approach are described in sections 3 to 8. Section 9 discusses related work. Section 10 consists of conclusions of our findings.

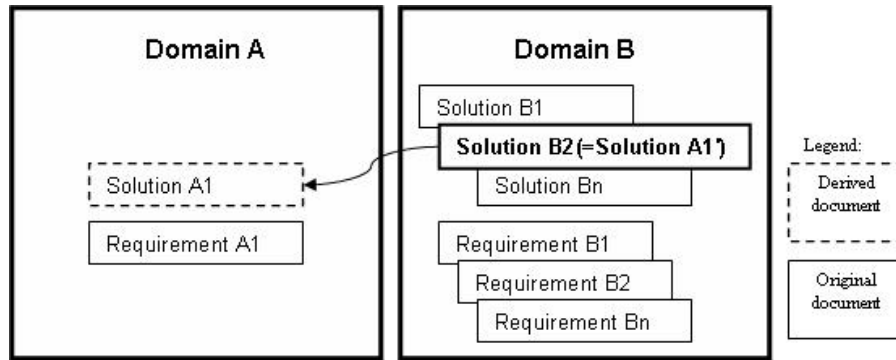


Figure 1: Cross-domain reuse of problem solutions

2 Case Study *Reuse of a sales department concept by the accounting department*

2.1 Requirements for the year end close in *Accounting*

The *Accounting* department of a company has collected its requirements for a new software system to automate the year end close processes. Figure 2 shows the corresponding requirements document. Based on this requirements document a solution should be provided. It is assumed that there is

Change Request #211
Subject: *Demands on the year end close*
Client: **Department for Accounting and Tax**
 The year end close process must fulfill the subsequent requirements:
 All assets such as e.g. current and non-current assets are considered during the fiscal year change. The stock at the end of the previous year is equal to the stock of the new year.

Figure 2: Requirements document created by the *Accounting* department

no specific solution available in the *Accounting* domain. So the search space is extended to other business management areas. It is assumed that a suitable solution exists in the business management area *Sales and Distribution*. The question is how this solution can be found without the specific expertise of this area, because the related documents are written in the special terminology of that area.

2.2 Requirements for the invoicing process in *Sales and Distribution*

According to the case study a solution document from another business area must fulfil the demands of the *Accounting* area. Using our experience and intuition we find an analogy within the business area *Sales and Distribution*. We construct a requirements document and a related solution document of this business area knowing that parts of the latter can be reused as a solution of the requirements in *Accounting*. The part of the solution that is regarded as reusable is the amount carried forward which is used within invoices if they contain at least two pages. The department for *Sales and Distribution* created the requirements document shown in Figure 3.

Change Request #116**Subject:** *Requirements concerning the invoicing process***Client:** Department Sales and Distribution (S&D)

The charging process should comply with the following:

The clarification process between the central charging department and the salesperson is very expensive and therefore not acceptable. Due to this fact the invoices should always be created by the salesperson who is responsible for the related order. An invoice should fulfill the following rules: To remain clear and readable the number of shown items per page is restricted to 10. If more than 10 items exist a new page is used. Nevertheless, each page should consider all values of the preceeding pages. At the end of each invoice a total sum is shown.

Figure 3: Requirements document created by the *Sales and Distribution* department

3 Methods to describe and model domain-specific problems and their solutions

Up to now two requirements documents from different business areas exist. We assume that at least parts of the solution for the second requirements document can be reused for the first one. Conversely, this means that also the requirements must be at least partly the same. Unfortunately, we face the problem that the corresponding documents are hardly comparable. On the one hand they have a quite inconvenient format as they are in unstructured full text, on the other hand they are written in their specific terminology in a specific context, hence the expressions are not very well comparable. Therefore, the challenge is to make the contents of the documents comparable in spite of the above mentioned situation and in spite of the fact that the developers only work on one specific project at a time and don't want to look at documents of another business area which only potentially contain reusable solutions. In practice the usual procedure is that the developers start to model a solution derived directly from the requirements document. In contrast to this procedure we propose that the developers start their problem solving procedure by splitting the requirements document into manageable pieces, which reflect the different problems and their specific contexts. We call such a specific problem description in its specific context a *Problem-Context-Description*. We illustrate this concept by picking one problem and its context out from the document.

3.1 Problem-Context-Analysis

The first step of the problem-context-analysis consists of identifying the different problems and their specific contexts. This is done by going through the requirements document sentence by sentence. This approach is comparable with identifying domain classes according to Rumbaugh et al. [RBP⁺91] which also supports the analysis of the problem and its context. The result is an assignment table with one column containing parts of the text, one column containing identified domain classes and one column containing a problem and context pair. First we analyze the requirements document for the invoicing process. Among others we can identify the following assignment of sentences and problem-context-description, see Table 1. In the following we will call this problem-context-description *invoice problem*. We selected this problem because it is the candidate which is suitable for the cross-domain reuse as we will show later. We apply the same method to the requirements document of the *Accounting* department for the year end close processes. One finding is the following problem-context-description and its sources, as illustrated in Table 2. In the following we will call this problem-context-description *year end close problem*.

3.2 Problem-Context-Modeling

After the analysis of the requirements and identifying the contained problems and their contexts we start to model them with UML class diagrams. Core of this approach is the rule that the classes and their relationships form the context and that the problem is modeled as a short textual statement.

Sentence in requirements document	Classes, Actors	Problem/Context
...
If more than 10 items exist a new page is used. Nevertheless, each page should consider all values of the preceding pages.	Item, Page	<i>Problem:</i> Values of preceding pages must be considered while the number of items per page is restricted. <i>Context:</i> Invoice has at least 2 pages.
...

Table 1: Assignment table of sentences and problem-context-description (invoice in *Sales and Distribution*)

Sentence in requirements document	Classes, Actors	Problem/Context
...
All assets like e.g. current and non-current assets are recognized during the fiscal year change. The stock at the end of the previous year is equal to the stock of the new year.	Assets, current assets, non-current assets, fiscal year, stock, previous year, new year	<i>Problem:</i> Previous year asset values not considered in new year. <i>Context:</i> Fiscal year change in balancing.
..

Table 2: Assignment table of sentences and problem-context-description (year end close in *Accounting*)

This statement is linked to the model element that causes the problem. We call this statement *problem statement*. Based on the problem-context-description *invoice problem* the problem-context-model can be created, see Figure 4. The cause of the problem is the cardinality. If the invoice contains more than one page then the problem exists. Due to this observation the problem statement is assigned to this model element. We have modeled the actual situation which is considered to be not satisfactory as it contains the problem *Account values of previous pages not considered*. The aim is to find a solution which doesn't contain the problem statement any more, so it has to be eliminated. The solution of the problem is the generally well-known *Carry Forward Account*. So the model is modified in the way that we differentiate the class *Account* between *Single Account* and *Carry Forward Account*. We obtain the class model of the solution according to Figure 5.

Note that this model doesn't contain the problem statement any more. It has been eliminated with this solution approach. The cardinality can be illustrated by the following examples: The invoice contains at most 10 single accounts. Then only one page and no carry forward account is needed. The invoice contains more than 10 but at most 20 single accounts. Then two pages and



Figure 4: Problem-Context-model for the invoice problem

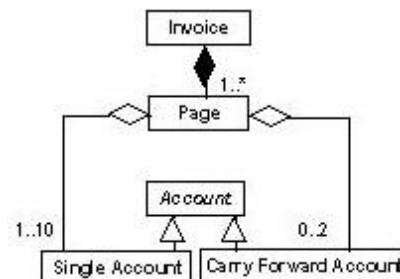


Figure 5: Solution model for the invoice problem

two carry forward accounts are needed. The problem-context-model for the *year end close problem* is shown in Figure 6.

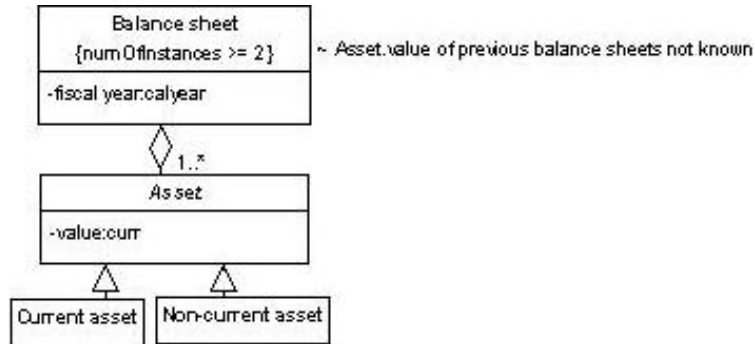


Figure 6: Problem-Context-model for the year end close problem

4 Creating analysis patterns: Method of subsequent abstraction

Until now we created problem-context- and solution models in the specific domains. But to be able to find a solution in another domain we must generalize these models. This method is characterized by the following ideas: The domain-specific models are abstracted in a two-step approach to make them comparable. The solution providing domain abstracts both the problem-context-model and also the related solution model. In the first step those model elements are eliminated that are not essential for the problem or its solution. In the second step the domain-specific terms are replaced by general terms. We call the generic instantiable representation of the problem-context-model *problem-context pattern*, the related solution *solution pattern*. We can summarize this approach with the roadmap shown in Figure 7. The solution seeking domain only abstracts its domain-specific

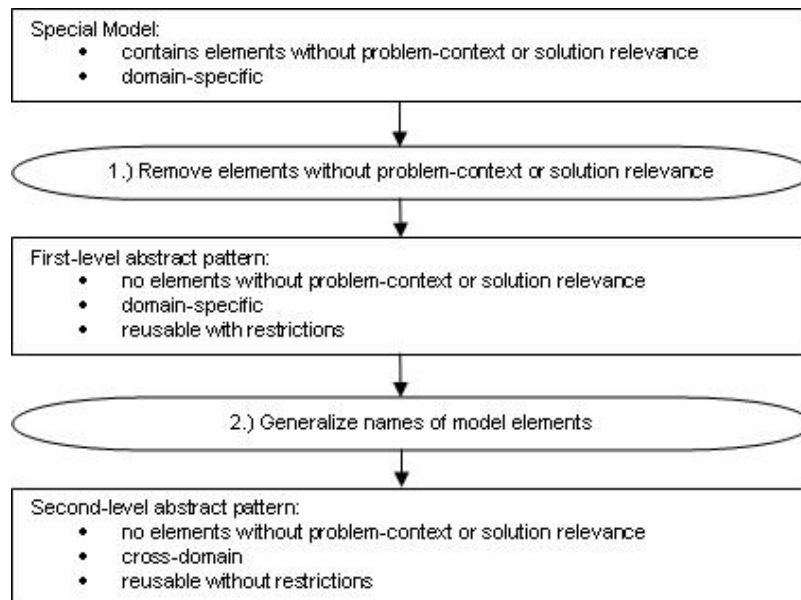


Figure 7: Abstraction roadmap: Principle of the two step abstraction

problem-context model as there's no solution available yet. This approach enables the solution seeking domain to use the problem-context pattern as a search key for comparable problem-context

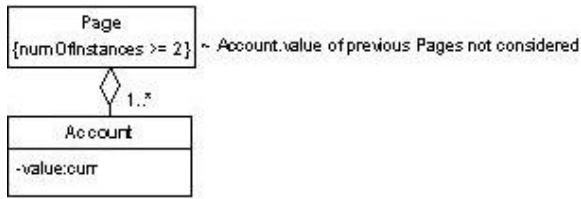


Figure 8: First-level problem-context pattern for the invoice problem

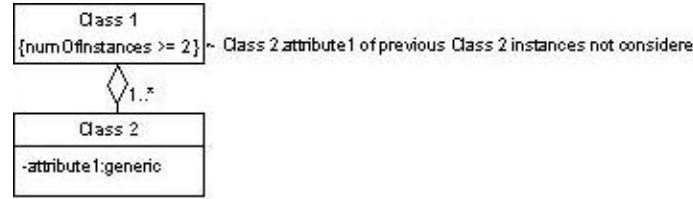


Figure 9: Second-level problem-context pattern for the invoice problem

patterns and consequently also for the related solution patterns in the solution providing domain. After having found a suitable solution pattern the solution seeking domain must instantiate this pattern. The abstraction is done in two steps reflecting the two operations which are applied to the models. The first operation removes inessential model elements, the second operation changes the domain-specific to cross-domain terms.

5 Patterns in the solution providing domain *Sales and Distribution*

We start with abstracting the problem-context-model for the *invoice problem*. In the first abstraction level all those elements are removed from the model that are not essential for the problem. The result of this process step is the *first-level problem-context pattern* shown in Figure 8. The class *Invoice* has been removed from the model as well as the aggregation relationship to the class *Page*, as they are not relevant for the problem itself. Only the classes *Page* and *Account* are affected by the problem. Nevertheless, the aggregation relationship between the class *Invoice* and the class *Page* is of importance because the problem statement takes also the cardinality into account ($n > 1$). This fact is recognized by the class constraint ($numberOfInstances \geq 2$) of the class *Page*. With this modification also the statement concerning the cardinality could be eliminated. The cardinality of the class *Account* has also been modified by replacing the maximum value '10' of the domain-specific model with '*'. There must be at least one instance of the class *Account* to fulfill the constraints given by the problem. This first-level problem-context pattern is characterized by the fact that it still contains domain-specific terms. As the aim is cross-domain reuse, a further abstraction step is performed that eliminates the domain-specific terms. The tags of the model elements are generalized in this step. Its result is the second-level problem-context pattern shown in Figure 9.

All class names have been replaced by the general tag *Class* and were numbered from top to bottom. The type of attributes has been replaced with *generic* as it has no relevance for the problem here. The next piece of work is the first-level solution pattern. After having performed the method of subsequent abstraction for the domain-specific problem-context-model we also apply it to the domain-specific solution model as shown in Figure 10. As in the problem-context-model the class *Invoice* is removed from the model and the constraint concerning the cardinality $n > 1$ of the relationship between this class and the class *Page* is replaced by a class constraint $numOfInstances \geq 2$. Furthermore, the cardinality of the class *Single Account* is changed from the value '10' to '*' as also here it is only important that there's at least one instance of the class *Single Account*. The result of the second abstraction step is the second-level solution pattern presented in Figure 11. The result of the pattern creation process is a pair of analysis patterns consisting of the problem-context pattern and the related solution pattern.

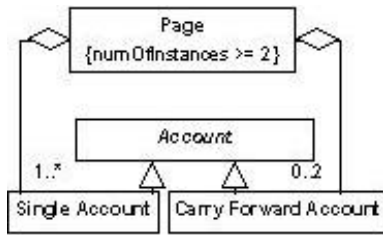


Figure 10: First-level solution pattern for the invoice problem

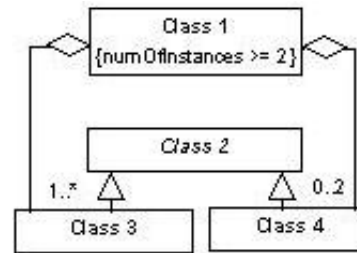


Figure 11: Second-level solution pattern for the invoice problem

6 Cross-domain problem-context pattern in the solution seeking domain *Accounting*

In the next step we apply the method of subsequent abstraction to the domain-specific problem-context-model *year end close problem* and get two problem-context patterns, see Figures 12 and 13.

While creating the first-level pattern we removed the attribute *fiscal year* from the class *Balance*

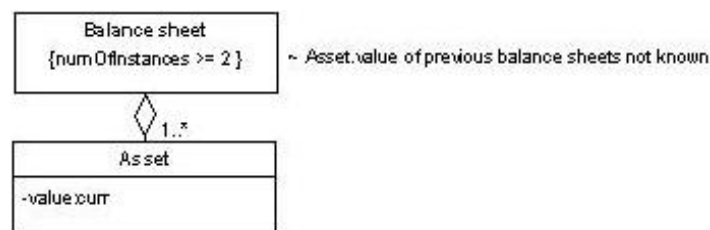


Figure 12: First-level problem-context pattern for the year end close problem

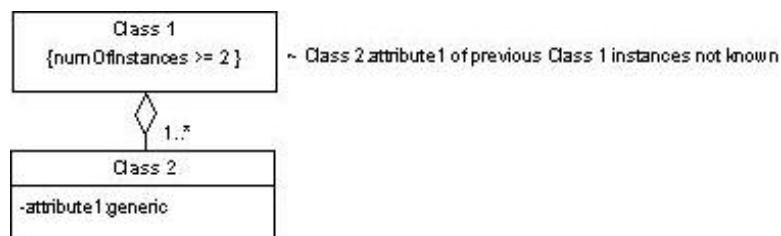


Figure 13: Second-level problem-context pattern for the year end close problem

sheet in the first abstraction step as it is not relevant for the problem statement. Furthermore, the classes *Non current asset* and *Current asset* are only presented by their superclass *Asset* as the described problem does not call for such a level of detail. In the second abstraction step all the terms were generalized.

7 Methods of searching for analysis patterns

Up to this point the solution providing domain (*Sales and Distribution*) as well as the solution seeking domain (*Accounting*) created second-level problem-context patterns. As already mentioned in Section 4 the problem-context pattern of the solution seeking domain is its search key to solutions in the other domain. Therefore, the solution of *Sales and Distribution* is usable for the *Accounting* department if both second-level problem-context patterns are congruent. Consequently the next step is a comparison between these two models. The class structure of the two problem-context patterns is nearly the same. In both patterns the classes *Class 1* and *Class 2* exist which have the same aggregation relationship as in both models *Class 1* aggregates *Class 2*. The problem statements of

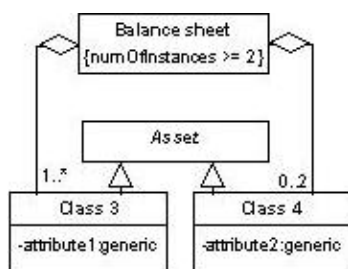


Figure 14: Solution model for the year end close problem (first approximation)

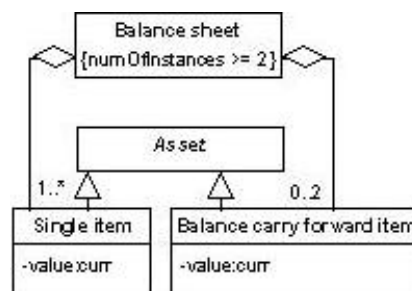


Figure 15: Solution model for the year end close problem (complete)

both problem-context patterns themselves make use of the expression *previous instances*. Moreover this expression is related to the class having assigned the problem statement (*Class 2*). Also the statements *not known* and *not considered* are similar enough, although there's some vagueness here that can result in different interpretations.

To sum up the two models are congruent, allowing a successful cross-domain search. Here we require that this cross-domain search is to some extent tolerant in the way that it can match terms with the same semantics.

8 Instantiation of the second-level solution pattern

After having found the second-level solution pattern it must be instantiated in the specific *Accounting* domain. First exactly those model elements are replaced which can be immediately derived due to the congruent expressions known from the first-level problem-context patterns. The result of this step is as shown in Figure 14. In this resulting solution model class *Class 1* has been replaced by class *Balance sheet* and the class *Class 2* is replaced by the class *Asset*. The more specific classes *Class 3* and *Class 4* are just left unchanged for the time being. They are not known from the first-level problem-context patterns as their introduction is part of the solution. The generic expressions of these classes can be replaced by terms that are known in the specific domain. Alternatively, if no suitable term exists, it has to be created by the domain. In this example the first option can be applied. *Class 3* is replaced by the term *Single item* and *Class 4* is replaced by the term *Balance carry forward item*. As the assets are always valued with currencies the attributes *attribute1* and *attribute2* can be replaced with a value and their type can be changed from *generic* to *curr*. The complete result of instantiating the second-level problem-context pattern is shown in Figure 15. Completing this step, the solution for the year end close process has been found and completely described. Even though it is very much simplified this solution is indeed used in accounting practice.

9 Related Work

The focus of our work is to provide a set of *methods* to create, search and reuse *cross-domain analysis patterns*. Part of the concept is that domain experts and developers should not need to gain cross-domain experience themselves to benefit from experience of domains they are not familiar with. Different from our method approach many contributions such as those of Fowler [Fow98], Fernandez and Yuan [FY01] or Sorgente, Fernandez and Petrie [SFP04] provide "concrete" analysis patterns which reflect the wide experience from projects across different domains that the writers themselves have gained or that they have collected. In fact, they represent *catalogues* of analysis patterns.

Another objective of Fernandez and Yuan [FY00] is to improve analysis modeling. In contrast, our goal is not to develop high-quality analysis models but to re-use conceptual problem solutions across expert domains in such a way that the real world is not only modeled, but improved. Fernandez and Yuan use an approach of abstraction and analogy to achieve their objective which, at a first

glance, seems to be quite similar to our approach, because it also considers cross-domain aspects. But there are some differences. First, their method relies on human comprehension of the given requirement to create an abstract pattern. In contrast, we try to define abstraction and matching of patterns in such a way that this process becomes amenable to tool support and (at least partial) automation. Hence, our abstract patterns are more generic. Second, we use so-called *problem statements*, i.e. we explicitly represent the aspect of a model that is improved by the pattern under consideration. We compare "before" and "after" states.

Purao and Storey [PS97] have developed a method to synthesize class models from natural language requirements descriptions. That synthesis is achieved by searching a pattern base and appropriately instantiating and combining the patterns contained in the pattern base. Their search technique is text-based and uses techniques from artificial intelligence, such as natural language processing, automated reasoning, and machine learning. In contrast, our approach is based on searching for *structural* similarities between different class models instead of analyzing textual descriptions. Moreover, Purao and Storey's methodology does not consider dynamic aspects, which we intend to take into account in further work. However, their approach to synthesize class models is potentially valuable for us and may be used to enhance our concept in future.

In a later paper [HPS99] Han, Purao and Storey go one step further and store *design fragments* instead of patterns in their knowledge base. Design fragments are larger entities than patterns, and they address specific sets of requirements. Design fragments are obtained by clustering of existing designs, where the clustering algorithm makes use of semantic similarities between the different designs. Such semantic similarities are very important also for the kind of reuse we have in mind, and they provide a promising idea to be included in the matching algorithm we need to determine if two problems match.

Other contributions focus on domain analysis without discussing cross-domain reusability of analysis patterns. Geyer-Schulz and Hahsler [GSH02] e.g. propose an outline template for analysis patterns that strongly supports the whole analysis process from the requirements analysis to the analysis model and further on to its transformation into a flexible and reusable design and implementation.

The work of Guerrieri [Gue95] deals with the question what can be done to institutionalize software reuse, especially in the area of domain analysis. It is pointed out that there is a need to standardize the representation of and to provide a mechanism to disseminate domain knowledge.

Another advantage of our method approach concerns traceability. Hamza and Fayad [HF04] point out that it is hard to satisfy both generality and traceability if analysis patterns are used as templates to develop a system. They say that once analysis patterns have been instantiated there is no link from the instance models to the analysis patterns anymore. In our concept such links between the analysis patterns and the instantiated models are crucial. Otherwise it would not be possible to realize the core mechanism of the concept consisting of the two-step abstraction and instantiation method.

10 Conclusions

In this paper, we have introduced a set of methods to systematically create and use cross-domain analysis patterns. We propose to describe the domain-specific problem in its appropriate context with a *problem-context-description* and transform it into a *problem-context-model*. If a solution is available then it will be represented as a *solution model*. Applying the *two-step abstraction method* to both models leads to a pair of related patterns consisting of a *second-level problem-context pattern* and a *second-level solution pattern*. Using a case study from the business world, we demonstrated that this mechanism works for *structural models*.

In our future work we will also try to apply these mechanisms to *dynamic models*. Besides this, we are confronted with open issues concerning the *problem statement* located in the problem-context pattern. As already indicated in Section 7 there is some vagueness concerning the expressions used in problem statements. To avoid this, some kind of consistency checks have to be performed to get a valid problem statement. Last but not least we want to prove that the output of our methods is compliant with existing analysis patterns.

References

- [Fow98] Martin Fowler. *Analysis patterns*. Addison-Wesley, 1998.
- [FY00] Eduardo B. Fernandez and Xiaohong Yuan. Semantic analysis patterns. In *ER*, volume 1920 of *Lecture Notes in Computer Science*, pages 183–195. Springer, 2000.
- [FY01] Eduardo B. Fernandez and Xiaohong Yuan. An analysis pattern for repair of an entity. In *8th Conf. on Pattern Languages of Programs (PLoP2001)*, Monticello, Illinois, USA, 2001.
- [GSH02] Andreas Geyer-Schulz and Michael Hahsler. Software reuse with analysis patterns. In *Proceedings of the 8th AMCIS*, pages 1156–1165. Association for Information Systems, Dallas, TX, 2002.
- [Gue95] Ernesto Guerreri. Enhancing the use of domain analysis. In *Proceedings of the Seventh Workshop on Institutionalizing Software Reuse*, 1995.
- [HF04] Haitham Hamza and Mohamed E. Fayad. Applying analysis patterns through analogy: Problems and solutions. *Journal of Object Technology*, 3(4), 2004.
- [HPS99] Tae-Dong Han, Sandeep Puro, and Veda C. Storey. A methodology for building a repository of object-oriented design fragments. In Jacky Akoka, Mokrane Bouzeghoub, Isabelle Comyn-Wattiau, and Elisabeth Mtais, editors, *Conceptual Modeling - ER '99, 18th International Conference on Conceptual Modeling, Paris, France, November 1999, Proceedings*, volume 1728 of *Lecture Notes in Computer Science*, pages 203–217. Springer, 1999.
- [PS97] Sandeep Puro and Veda C. Storey. Intelligent support for retrieval and synthesis of patterns for object-oriented design. In David W. Embley and Robert C. Goldstein, editors, *Conceptual Modeling - ER '97, 16th International Conference on Conceptual Modeling, Los Angeles, California, USA, November 3-5, 1997, Proceedings*, volume 1331 of *Lecture Notes in Computer Science*, pages 30–42. Springer, 1997.
- [RBP⁺91] James E. Rumbaugh, Michael R. Blaha, William J. Premerlani, Frederick Eddy, and William E. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.
- [SFP04] Tami Sorgente, Eduardo B. Fernandez, and Maria M. Larrondo Petrie. Analysis patterns for patient treatment. In *8th Conf. on Pattern Languages of Programs (PLoP2004)*, Monticello, Illinois, USA, 2004.
- [TZZ00] John Terninko, Alla Zusman, and Boris Zlotin. *Step-by-Step TRIZ Creating Innovative Solution Concepts*. Verlag Moderne Industrie, 2000.