Problem-Oriented Documentation of Design Patterns

Alexander Fülleborn[1], Klaus Meffert[2] and Maritta Heisel[1]

[1] University Duisburg-Essen, Germany
alexanderfuelleborn@hotmail.de
maritta.heisel@uni-duisburg-essen.de
[2] Technical University Ilmenau, Germany
pattern@klaus-meffert.com

Abstract. In order to retrieve, select and apply design patterns in a tool-supported way, we suggest to construct and document a *problem-context pattern* that reflects the essence of the problems that the design pattern is meant to solve. In our approach, software engineers can choose examples of source code or UML models from the special domains that they are experts in. We present a method that enables software engineers to describe the transformation from a problem-bearing source model to an appropriate solution model. Afterwards, the *inverse* of that transformation is applied to the UML solution model of the existing design pattern, resulting in an abstract problem-context pattern. This pattern can then be stored together with the solution pattern in a pattern library. The method is illustrated by deriving a problem-context pattern for the Observer design pattern.

Key words: Design patterns, problem derivation, model abstraction, cross-domain documentation, problem-context patterns, UML models, source code

1 Introduction

Design patterns support software engineers in creating maintainable and extendable software. In order to select and apply design patterns, practitioners typically learn a pattern by reading a design pattern book or paper or by studying UML diagrams or source codes, respectively. From our point of view, this kind of documenting design patterns is lacking a machine-processable representation of the *problems* in their contexts to be solved by the design patterns. The pattern itself as the solution for the related problems, however, is represented by UML models, besides explanations in natural language and program code examples. This is why it can be instantiated as a solution model to the concrete problems. As there exists no corresponding problem model, it is difficult for software engineers to judge whether their concrete source code or design models, respectively, match the problems that the design pattern is meant to solve. They are forced to make a comparison that is based on two different formats in order to select an

appropriate pattern. In addition, the contexts of the problems are expressed on different abstraction levels. Due to the lack of cross-domain knowledge, there is a risk that software engineers assume their problem to be of a domain-specific type that does not match the essence of the problem addressed by a design pattern. Therefore they do not choose the solution provided by such a design pattern, even if it solved their problem.

We are interested in developing methods that help software engineers to retrieve, select and apply design patterns in a tool-supported way. It should be possible to apply these methods in the forward as well as in the re-engineering phase of the software product lifecycle. In this paper, we put special emphasis on *problem orientation* in documenting design patterns. We illustrate our ideas by a re-engineering example. Key of our approach is to enable software engineers in their role of pattern documentalists to use their daily, domain-specific work together with their knowledge about design patterns, in order to complete the documentation of these design patterns. We introduce the possibility for software engineers to start this completion process either with UML models or with source code, in order to obtain appropriate UML models that reflect the essence of the related problems in their contexts. The resulting artefacts that we call *problem-context patterns* are the basis for our overall methodology of semi-automated retrieval, selection and application of design patterns.

Our approach consists of a sophisticated way of documenting the situation before and after a design pattern is being applied. For the first part, this documenting is done by adding non-functional requirements as annotations to concrete, domain-specific source code or UML models that have design deficiencies, in order to document the problems in their contexts that the chosen design pattern solves. For the second part, we formally document the solved problems in a way that they can be compared to the situation before the chosen design pattern was applied. By way of that comparison, the *transformation* between the situation before and after applying the design pattern is made explicit. This transformation is then reused on the design pattern abstraction level in order to derive the reusable cross-domain representation of the situation before the chosen design pattern is being applied. To obtain the problem-context pattern, the *inverse* of the transformation is applied to the already existing UML model of the chosen design pattern that we call *solution pattern*.

The rest of the paper is organized as follows: in Section 2, we introduce our method for deriving problem-context patterns. We illustrate our method in Section 3 by using an example from the business domain of *human resources* and the Observer design pattern. Section 4 discusses other work in this area. Section 5 consists of conclusions of our findings and an outlook on future work.

2 A Method for Deriving Problem-Context Patterns

An overview of our method is given in Table 1.

Table 1. Method for deriving problem-context patterns

Step	Description
1	Choose a problem-bearing, domain-specific source code or UML model example
2	Annotate the chosen problem-bearing source code and UML models with *problem motives*
3	Perform transformations by applying design pattern under consideration to the chosen source code and UML models
4	Annotate the resulting source code and UML models with *solution motives*
5	Annotate the UML solution model of the cross-domain design pattern with the same solution motives as in Step 4
6	Perform *inverse* design pattern transformations to the existing design pattern UML solution models that are annotated according to Step 5

Steps 1 to 4 are performed on the domain-specific level. In Step 1, software engineers choose a concrete, problem-bearing source code or UML model example that exists in their specific expert domain. In case software engineers choose a problem-bearing source code for which no corresponding UML model exists yet (the typical re-engineering case), the latter must be created, as it is needed in the later steps of the method. In case software engineers choose a problem-bearing UML model as a starting point for the example (the typical forward engineering case), they do not need to have corresponding source code, as Steps 5 and 6 are only based on the UML models. The chosen example must fit to the abstract, natural-language problem description of the design pattern for which they want to complete the documentation. We assume that software engineers are familiar with the design pattern, for which they intend to complete the documentation. By using specific examples from expert domains, the procedure of completing a design pattern documentation is facilitated. The advantage of this approach is that it is integrated into the usual work of software engineers. The effort needed to complete the documentation of design patterns is minimized, because software engineers can derive them by doing their daily work of modeling, coding and improving designs.

In Step 2, annotations to the problem-bearing source code and to the UML models are added manually. We call these annotations *problem motives*. A method for deriving problem motives on the source code level can be found in [5]. On the modeling level, we assign these problem motives to UML model elements. They represent the non-functional requirements that need to be fulfilled by the source code and UML models after the design pattern was applied.

In Step 3, the problem-bearing source code and UML models are transformed step by step to re-engineered new source code and UML models, according to the knowledge embodied in the design pattern. In case new elements are added or existing elements are changed, annotations are manually added to these elements in Step 4. We call these annotations *solution motives*. They directly correspond

to the problem motives in the problem-bearing source code and UML models. It is also possible that there does not exist a problem motive that corresponds to a solution motive. In this case, the solution provides an additional advantage. It is also possible that a problem motive without any corresponding solution motive exists. This indicates a non-optimal solution of the problem.

The preceding steps all take place on the domain-specific level with its special vocabulary and semantics, and with limited reuse potential. In the final two steps of our method, software engineers operate on the generic, cross-domain design pattern level. Here, the main purpose is to *complete the documentation of the generic design pattern* that can be reused across domains. The goal is to find an appropriate cross-domain UML model for the situation before transformations are being performed.

In Step 5, software engineers annotate the UML solution model of the chosen design pattern that has been applied on the domain-specific level before. For this purpose, they reuse the knowledge they already gained in the domain-specific scenario: they take the same solution motives they also used for the domain-specific UML solution model and add them as annotations to the cross-domain UML solution models. Then, they also reuse the transformations, but in this case, they apply these transformations *inversely* to the generic, cross-domain UML solution models of this design pattern. Thus, they *derive the cross-domain problems in their cross-domain contexts* that fit to the design pattern. Finally, the obtained problem-context pattern can be stored together with the solution pattern in a design pattern library. It then can be retrieved and selected according to the method described in [2].

3 Case study *Salary Statement Application*

To illustrate our method, we present a case study from the *human resources* business domain. It is about an existing software application for creating *salary statements* of monthly employee salaries. This application needs to be re-engineered, as it has non-functional deficiencies that can be removed by applying the Observer design pattern. The software engineers who perform this re-engineering task know the pattern well and are able to apply it to the existing code. No UML models exist, only source code is available to them. Hence, this source code represents the problem-bearing, domain-specific source code example according to Step 1 of our method. Besides this domain-specific, pure re-engineering task, the software engineers are also asked to complete the Observer design pattern documentation. Hence, they are asked to perform the remaining steps of our method. In Sections 3.1 and 3.2, we present the results of Steps 1 to 4 of our method on the source code level. We chose the object-oriented Java language for demonstrating purposes as Java is widely spread and state-of-the-art. In Sections 3.3 and 3.4, we illustrate the results of Steps 1 to 4 on the UML model level. In Section 3.5, the results of Steps 5 and 6 are presented.

3.1 Salary statement application: annotated problem-bearing, domain-specific source code

According to Steps 1 and 2 of our method, the following source code from the salary statement application has been chosen and annotated:

```
010       public class SalaryStatementApplication {
020         public static void main(String[] args) {
030           EmployeeDetail   employeeDet   = new EmployeeDetail();
040 acd       EmployeeSelector employeeSel = new EmployeeSelector(
    employeeDet);
050           // User Input in a GUI field. Sent by an event handler
060           employeeSel.updateEmployeeID("D026143");
070         }
080       }

100       public class EmployeeSelector {
110 a       private EmployeeDetail employeeDet;
120         private PersNo         selectedEmployee;

130 acd     public EmployeeSelector(EmployeeDetail employeeDet) {
140 ac        this.employeeDet = employeeDet;
150         }

180         // Method will be called after user entered employee ID
190         public void updateEmployeeID(PersNo ID) {
200           selectedEmployee = ID;
210 a         employeeDet.changeEmployeeID(ID);
220         }
230       }

300       public class EmployeeDetail {
340         public void changeEmployeeID(PersNo ID) {
350           // Read employee with given ID from database
360           PersName name = ...
370           // Read salary data for employee from database
380           Salarybracket salarybracket = readSalarybracket(ID);
390           // Update the display of the graphical user interface
400           ...
410         }

420         public Salarybracket readSalarybracket(PersNo ID) {
430           // Read salary data with given ID from database
440           Salarybracket result = ...
450           return result;
470         }
480       }
```

Listing 1. Salary statement application: annotated problem-bearing, domain-specific source code

The source code given in Listing 1 represents an application with three classes. The first class, *SalaryStatementApplication*, is responsible for starting the application. It creates two graphical elements. The first element is represented by the class *EmployeeSelector* and implements a selection list with basic master data related to employees. When users have chosen an entry from this list, the detail information that is needed in this context can be read and displayed. In our example, the detail information is represented by the second element, the class *EmployeeDetail*, and is displayed as a screen area with detail data such as *Name* and *Salarybracket* (which is the technical term for a salary group) of the employee, that has been selected by the employee selector. As already men-

tioned above, the Observer design pattern should be applied to this source code. According to the terminology used in Observer, the class *EmployeeSelector* corresponds to the *Subject* class, and *EmployeeDetail* is the observer class to the subject. The basic non-functional deficiency of this source code is the fact that the classes *EmployeeSelector* and *EmployeeDetail* are too tightly coupled, which is not desired. We can further differentiate this fundamental problem for the given context. Firstly, the subject class *EmployeeSelector* can only be reused in a limited way, as it has an attribute of type *EmployeeDetail*. A common interface provided by the subject class to share data or functionality with observers is missing. This is a structural coupling. Secondly, objects of the class *EmployeeDetail* can only be registered at one point in time, namely when objects of class *EmployeeSelector* are created. Thus, the objects of the classes *EmployeeDetail* and *EmployeeSelector* are also coupled in time. Thirdly, only objects of the class *EmployeeDetail* can be registered with the class *EmployeeSelector*. A unified registering mechanism for objects of arbitrary classes is missing, despite of the fact that they are not present at the moment. According to Step 2 of our method, the described detail problems are added to the source code as *problem motives* by adding a character as an identifier to each detail problem. In Table 2, the used problem motives are listed.

Table 2. Salary statement application: problem motives

Problem motive identifier	Description
a	Observer class does not implement any interface
c	Observer class can only be registered at one given point in time
d	No unified registering mechanism existing

Table 2 contains several aspects of the *too tightly coupled* problem, which are addressed by the Observer design pattern. Note that identifier *b* does not appear in the problem-bearing source code. The reason is that it has been reserved for a solution motive in the post-transformation source code, that we will discuss later. Performing Step 2 of our method, we place problem motive identifiers after the line numbers in the problem-bearing source code. As already mentioned before, a method exists for annotating source code which can be found in [5]. For example, problem motive *a* has been declared for all lines of code that are affected by the fact that class *EmployeeDetail* does not implement any interface. These are all lines of code where this class type is used. Applying the Observer design pattern removes the deficiencies that are mirrored by the problem motives. In the following, the Observer design pattern is applied to the given source code according to Steps 3 and 4 of our method.

3.2 Salary statement application: performing transformations by applying the Observer design pattern

According to Step 3 of our method, every change of the problem-bearing source code that is caused by applying the Observer design pattern is reflected by a transformation. Performing method Step 4, each of these transformations has an explanation, given by an annotated solution motive. This solution motive is the complement to the problem motive and is directly related to the latter. A problem motive that has been replaced by a solution motive indicates that the given problem is solved. A solution motive without any corresponding problem motive reflects a positive property that has been added without any problem relationship. This is true for solution motive *b*, that we already mentioned, and it is true for solution motive *e*. An identifier of a solution motive that equals an identifier of the problem motive indicates that it solves the corresponding problem. An overview of the solution motives used in this example is given in Table 3.

Table 3. Salary statement application: solution motives

Solution motive identifier	Description
a	Treat observer classes equally
b	Possibility to register any number of observers
c	Possibility to register at any time
d	Unified registering mechanism
e	Notify all observers

The transformations used to apply the Observer pattern are all related to these solution motives. For each transformation, we give the resulting source code including the solution motives. If the transformation is a deletion, only the deleted source code is shown. For better traceability, the line numbers from Listing 1 are given, too. Equal line numbers in the post-transformation source code indicate a change, new line numbers indicate an insertion.

Transformation T1: declaring the subject without relation to any observer
In this transformation, the observer and subject class are decoupled from each other by deleting the static attribute for observer from the subject class.

```
109   /**@@motive a(1): Treat observer classes equally*/
110 a      private EmployeeDetail employeeDet;
```

Transformation T2: constructing the subject without relation to any observer
This transformation causes the change of two dependent parts within the source code, namely of a constructor declaration and the creation of objects of class *EmployeeSelector* by this constructor. The changed constructor looks as follows:

```
127   /**@@motive a(2): Treat observer classes equally*/
128   /**@@motive c(1): Possibility to register at any time*/
129   /**@@motive d(1): Unified registering mechanism*/
130   public EmployeeSelector()
137   /**@@motiv a(3): Treat observer classes equally*/
138   /**@@motive c(2): Possibility to register at any time*/
139   /**@@motive d(2): Unified registering mechanism*/
140     this.employeeDet = employeeDet;
```

As a result, the constructor call must also be adjusted:

```
037     /**@@motive a(4): Treat observer classes equally*/
038     /**@@motive c(3): Possibility to register at any time*/
039     /**@@motive d(3): Unified registering mechanism*/
040     EmployeeSelector employeeSel = new EmployeeSelector();
```

As every solution motive can appear more than once, we use a serial number per motive, which we put into brackets behind each solution motive.

Transformation T3: introducing the base class for observer
Part of the core concept of the Observer design pattern is the usage of an abstract base class for observer classes. In this context, we call this class *Observer*. Firstly, we introduce this base class:

```
899 /**@@motive a(5): Treat observer classes equally*/
900 public abstract class Observer
909   /**@@motive e(1): Notify all observers*/
910   public abstract void update(Object state);
920
```

Next, the specialized observer class *EmployeeDetail* can inherit from this class:

```
299 /**@@motive a(6): Treat observer classes equally*/
300 public class EmployeeDetail extends Observer
```

Furthermore, the abstract method needs to be implemented:

```
309   /**@@motive e(2): Notify all observers*/
310   public void update(Object state)
320     changeEmployeeID((PersNo)state);
330
```

Transformation T4: introducing a universal registration mechanism within the subject
Now, we can introduce a registration method with a variable for storing observer references:

```
109   /**@@motive b(1): Possibility to register any number of observers*/
110   private List<Observer> observers = new Vector();
```

```
147   /**@@motive a(7): Treat observer classes equally*/
148   /**@@motive b(2): Possibility to register any number of observers*/
149   /**@@motive c(4): Possibility to register at any time*/
150   public void register(Observer a_observer)
160     observers.add(a_observer);
170
```

The newly introduced solution motive with the identifier *b* has no corresponding problem motive in the problem-bearing source code. This means that it is an additional advantage of the solution that was not seen as a problem before.

Transformation T5: unified notification of all observers by the subject

The way the subject notifies its observers can be adjusted, too. Firstly, we introduce a new method for notifications:

```
222   /**@@motive e(3): Notify all observers*/
223   public void notify()
224     /**@@motive a(8): Treat observer classes equally*/
225     /**@@motive e(4): Notify all observers*/
226     for(Observer observer:observers)
227       observer.update(selectedEmployee);
228
229
```

Now the method call must be placed at a suitable location:

```
209     /**@@motive e(5): Notify all observers*/
210     notify();
```

The remaining reference to *EmployeeDetail* in line 030 of the problem-bearing source code can be replaced by a reference to the class *Observer*. However, this is not needed here. To give readers of this paper a complete overview about the resulting post-transformation source code, we provide it as a listing in an appendix at the end of the long version of this paper.[1]

3.3 Salary statement application: annotated problem-bearing, domain-specific UML model

Until now we performed Steps 1 to 4 of our method on the source code level. Next, we repeat these steps on the UML model level, because Steps 5 and 6 are based on the UML models. We start with Steps 1 and 2. First, we create domain-specific UML models of the annotated problem-bearing source code. Extracting the model can be automated to a certain extent. Next, the problem motives of the source code are added to those model elements that cause the problem, according to Step 2 of our method. While taking over the problem motives, some information gets lost, as not all of the problem motives can be taken over to the UML model level. The reason is that problem motives in the source code can also relate to single program statements that do not appear on

[1] Available at http://swe.uni-duisburg-essen.de/techreports/Fase09Longversion.pdf

the UML model level. However, this loss of information can be reduced by using UML comments, which contain these program statements with assigned problem motives in order to give software engineers some guidance in implementing the details. The resulting UML problem-context model, which is needed later in Step 6 to derive an appropriate problem-context pattern, is shown in Figure 1.

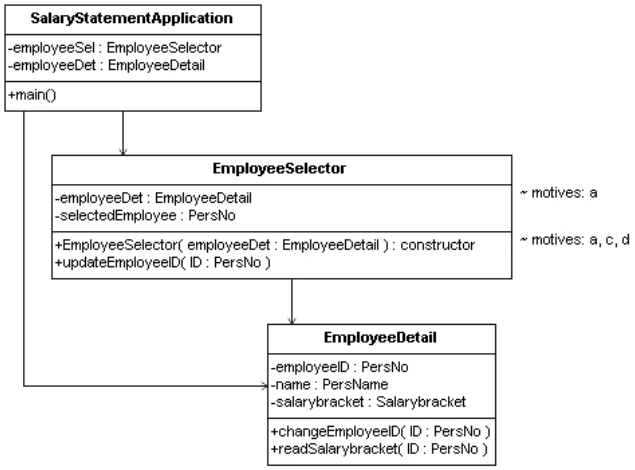

Fig. 1. Salary statement application: annotated UML model (problem-context model)

In this problem-context model, the problem motive identifiers of the annotated problem-bearing source code are assigned to the relevant elements using the tilde symbol. For example, annotation *motives: a* is assigned to the problem-causing attribute *employeeDet* in the class *EmployeeSelector*. As stated in Table 2, this identifier means *Observer class does not implement any interface*, which refines the basic problem that class *EmployeeSelector* and class *EmployeeDetail* are too tightly coupled. This problem is expressed by a static attribute that limits the reuse of class *EmployeeSelector*. For the sake of readability, we only look at the structural aspects in our example. In our research work, we also applied the described steps to sequence diagrams, which works well, but does not provide additional information and does not necessitate any extension of the method.

3.4 Salary statement application: annotated resulting domain-specific UML model

In this section, we repeat Step 2 of our method on the modeling level. The resulting domain-specific UML solution model is directly generated from the re-engineered source code. Then, software engineers analyze the differences between the pre- and post-transformation UML models by considering the differences in the pre- and post-transformation source codes, and annotate the models with

comments about the solved problem. The resulting annotated class diagram is shown in Figure 2.

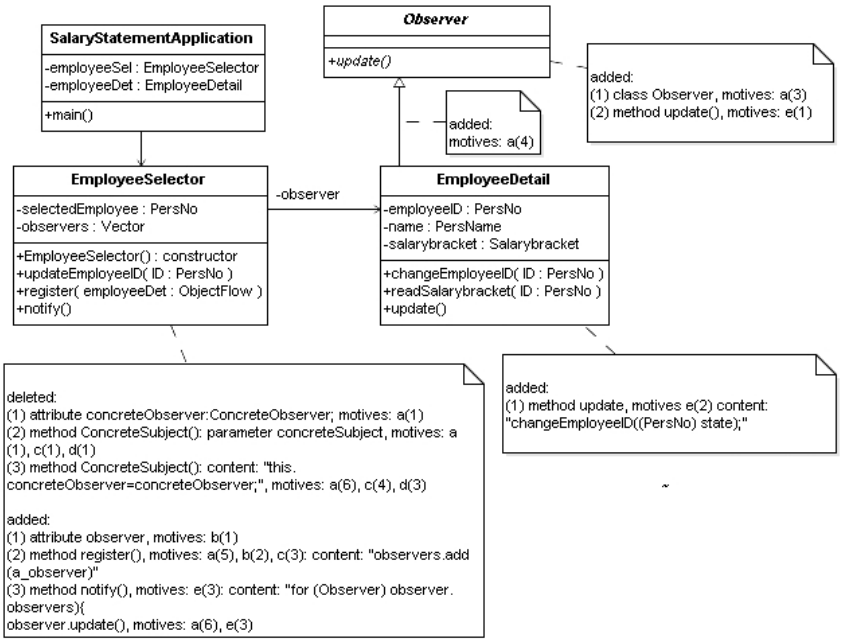

Fig. 2. Salary statement application: annotated resulting post-transformation UML solution model

Note that there are comments assigned to several model elements. These model elements are exactly the classes or relationships that have been changed from the problem-context model. The comments log the type and the reason for the change. The type of changes, namely *adding* or *deleting* is described by an appropriate keyword. The reason for the change is described by adding solution motives, which also establish relationships to the underlying problem motives in the problem-context model. For example, a comment is added to the class *EmployeeSelector* that states that the attribute *employeeDet* is deleted due to solution motive *a*. In the problem-context model, this attribute still exists and is annotated with the corresponding problem motive. By using a numbering within the used motives, software engineers can better reflect the order of the single transformation steps and the involved model elements.

3.5 Deriving a fitting problem-context pattern (cross-domain)

Up to this point in the procedure, all activities take place on the domain-specific level. From now on, the cross-domain level is considered by following Steps 5 and 6 of our method. Here, the main purpose is to *complete* the generic, cross-domain

Observer design pattern by finding an appropriate cross-domain UML model, a problem-context pattern for the pre-transformation situation. First, the existing UML solution model of the Observer design pattern is annotated according to Step 5 of our method. For this purpose, the solution motives that have been used to annotate the domain-specific *salary statement* UML solution model are taken as a starting point. Thus, software engineers extend the Observer UML solution models with information about the problems they solve and with information about the way how they solve them. As in the domain-specific example, for the sake of readability and simplicity, we use a variant of the Observer design pattern. In this variant, no explicit subject superclass exists, and observers do not ask separately for state changes. Instead, the subject provides the observers proactively with information about state changes. While adding information about the performed transformations, the transformed model elements are also abstracted. The result is illustrated in Figure 3.

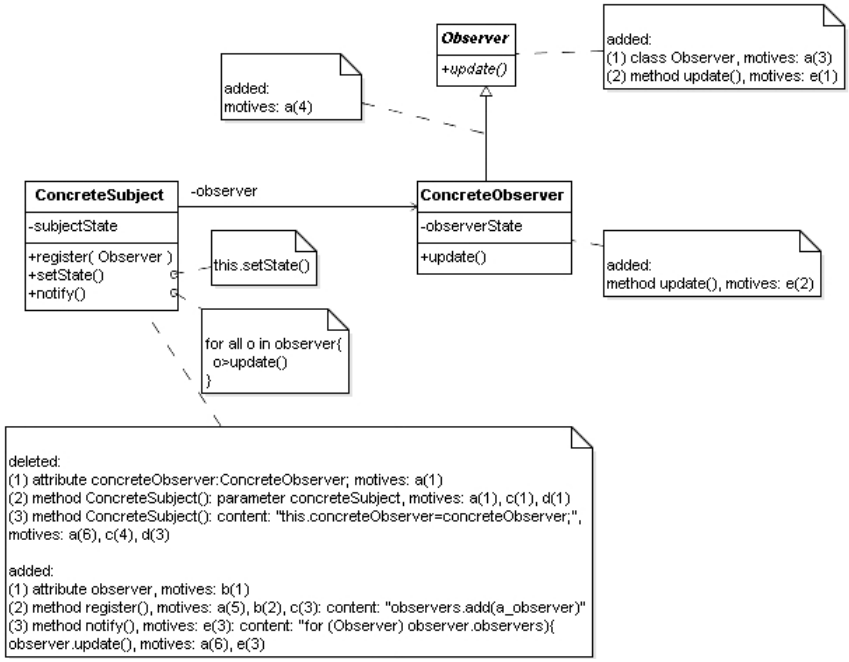

Fig. 3. Annotated UML solution model of the Observer design pattern (variant) according to method Step 5

Note that annotations have also been changed. They are adapted to the pattern in the sense of an abstraction. For example, the model element *attribute employeeDet* of class *ConcreteSubject* that can be found under the *deleted* annotation has been renamed to *attribute concreteObserver*. This annotated solution pattern is the starting point for the next step of *inverse transformations*. In Step 6 of our method, a suitable Observer problem-context pattern is derived by

applying inverse transformations. All transformation steps, that have been performed before on the domain-specific level, started from problem-bearing models (problem-context models) and resulted in solution models. On the cross-domain level, solution models exist already, as they are described in the literature [3]. Thus, in order to obtain appropriate problem models for the Observer design pattern, the knowledge about the way to transform the problem-bearing models into solution models on the domain-specific level is reused, but in the opposite direction. For example, in Figure 3 the class *Observer* is annotated with:

```
added:
(1) class  Observer, motives: a(3)
(2) method update(), motives: e(1)
```

This annotation is based on the transformations *add class Observer* and *add method update() to class Observer*. Thus, the appropriate inverse transformations are:

```
delete method update() from class Observer
delete class Observer
```

Besides performing inverse transformations, also problem motives are needed in the derived problem-context pattern, which establish the link to the solution motives in the solution pattern. These problem motives are derived from the domain-specific problem-context model. The derived *problem-context pattern* is illustrated as a result of the described actions in Figure 4.

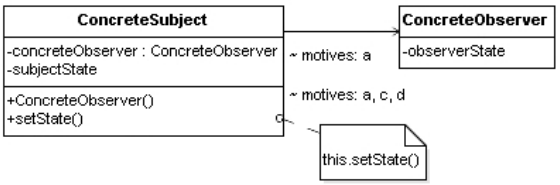

Fig. 4. Derived problem-context pattern of the Observer design pattern according to method Step 6

The illustrated problem-context pattern abstractly describes a possible starting point for applying the Observer design pattern, using the same means of expression as the solution, namely UML. Together with the UML solution models, the *solution pattern*, it forms the Observer design pattern and can be stored in a pattern library. The problem-context pattern is the access key for a semi-automated pattern retrieval and selection method. Such a method is described in [2].

4 Related Work

There has only been little work on documenting the problem essence of design patterns in an appropriate way. Different from our approach of expressing the problem essence as UML models, other contributors take *UML meta models* as the basis for their methods. Mili and El-Boussaidi [6] use transformation meta

models, besides problem and solution meta models, to describe appropriate design pattern problem models and the way they are transformed to design pattern solution models. Differently from our method, their problem meta models do not contain any non-functional requirement or problem description that would be comparable to our problem motives.

Kim and El Khawand [4] propose to rigorously specify the problem domain of design patterns. In contrast to Mili and El-Boussaidi [6] and our approach, they do not describe the problem-bearing model *before* a design pattern is applied to it. Moreover, they focus on the functional aspects of design patterns, not on non-functional aspects. The authors are mainly interested in developing tool support for checking whether existing UML models conform to known design patterns.

The work of Fanjiang and Kuo [1] introduces the concept of design-pattern-specific *transformation rule schemata* to be used as an additional design pattern documentation. The transformation steps, which are described in natural language, are similar to ours and are helpful in applying design patterns. However, as the authors do not intend to support software engineers in their role of documentalists, they do not give guidance on deriving transformation rule schemata.

The work of O'Cinnéide and Nixon [7] aims at applying design patterns to existing legacy code in a highly automated way. They target code refactorings. Their approach is based on a semi-formal description of the transformations themselves, needed in order to make the changes in the code happen. In contrast to our method, they describe precisely the transformation itself and under which pre- and postconditions it can successfully be applied. In our work, we illustrate the situation before and after the transformation. To a certain extent, the described preconditions of the transformations can be compared with our problem context as it outlines the situation before the design pattern is applied. The advantage of our approach, however, is that we *explicitly* describe the non-functional deficiencies by using annotations in the source code of the sub-optimal situation.

5 Summary and Future Work

In this paper, we have presented a method for the problem-oriented documentation of design patterns that consists of 6 steps. The results of Steps 1 to 2 are an annotated problem-bearing source code and the corresponding UML models, stemming from the practical work of software engineers. By applying the chosen design pattern and by adding *solution motives* to the resulting source code and corresponding UML models, the outcome of Steps 3 and 4 are the transformed and annotated solution source code and UML models. In Step 5, the same solution motives are used to obtain annotated UML solution models of the cross-domain design pattern. Finally, inverses of transformations according to Step 3 are applied to these UML solution models, which result in a *problem-context pattern* that fits to the chosen design pattern.

The use of expert domain, real world examples on the source code or on the modeling level in order to derive the problem-context pattern on the cross-domain level is novel and makes this approach especially useful and efficient in re-engineering as well as in forward engineering projects. In addition to that reuse of

domain-specific knowledge on the abstract level, reusing the already documented UML models of the design pattern solution part to derive the abstract problem-context pattern is efficient.

To demonstrate how our method works, we used Observer, a behavioural design pattern. In our research work, we also applied it to creational and structural patterns, which works well, too. Besides the scenario of completing *existing* design patterns that are known from the literature, it also seems promising to support the creation of *new* patterns in this way. This aspect is part of our future work. Furthermore, we are working on the development of a *problem statement language* that helps to reuse generic, standardized problems by making use of the ideas provided by Willms et al. [8]. Another subject area we want to address is the cross-domain reuse of functional requirements as opposed to the typical non-functional requirements that are addressed by design patterns.

References

1. Yong-Yi Fanjiang and Jong-Yih Kuo. A pattern-based model transformation approach to enhance design quality. In H.D. Cheng, S.D. Chen, and R.Y. Lin, editors, *JCIS 2006, Proceedings of the 2006 Joint Conference on Information Sciences*. Atlantis Press, 2006.
2. Alexander Fülleborn and Maritta Heisel. Methods to create and use cross-domain analysis patterns. In Uwe Zdun and Lise Hvatum, editors, *EuroPLoP '06, Proceedings of the 11th European Conference on Pattern Languages of Programs*, pages 427–442. Universitätsverlag Konstanz, 2007.
3. Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In Oscar Nierstrasz, editor, *ECOOP'93 - Object-Oriented Programming, 7th European Conference*, pages 406–431. Springer, 1993.
4. Dae-Kyoo Kim and Charbel El Khawand. An approach to precisely specifying the problem domain of design patterns. *Journal of Visual Languages and Computing*, 18(6):560–591, 2007.
5. Klaus Meffert and Ilka Philippow. Supporting program comprehension for refactoring-operations with annotations. In Hamido Fujita and Mohamed Mejri, editors, *Proceedings of the fifth SoMeT06: New Trends in Software Methodologies, Tools and Techniques*, volume 147, pages 48–67. IOS Press, 2006.
6. Hafedh Mili and Ghizlane El-Boussaidi. Representing and applying design patterns: What is the problem? In Lionel C. Briand and Clay Williams, editors, *MoDELS 2005, Model Driven Engineering Languages and Systems, 8th International Conference*, pages 186–200. Springer, 2005.
7. M. O'Cinnéide and P. Nixon. A methodology for the automated introduction of design patterns. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, page 463, Washington, DC, USA, 1999. IEEE Computer Society.
8. Janine Willms, Ina Wentzlaff, and Markus Specker. Kreativität in der informatik: Anwendungsbeispiele der innovativen prinzipien aus triz. In *Informatik 2000, Neue Horizonte im neuen Jahrhundert, 30. Jahrestagung der Gesellschaft für Informatik*. Springer Berlin / Heidelberg, 2000.