Computing and Informatics, Vol. 36, 2017, 55–85, doi: 10.4149/cai_2017_1_55

DESIGN PATTERN INSTANCES WITHIN MODEL DRIVEN DEVELOPMENT BASED ON ABSTRACTION, CONCRETIZATION AND VARIABILITY

Peter Lacko, Peter Kajsa, Pavol Návrat

Faculty of Informatics and Information Technologies
Slovak University of Technology
Ilkovičova 2, 842 16 Bratislava, Slovakia
e-mail: {lacko, kajsa, navrat}@fiit.stuba.sk

Abstract. The main goal of the paper is to present the method of design pattern support based on principles of model driven development: the abstraction, semantics and model transformations. More specifically, the method is based on the principle of suggestion of design pattern instances via the semantic marking of model elements or source code fragments and on the subsequent transformations of this way marked models or source code. Thanks to the continual support of the design patterns at more levels of abstraction and thanks to the transformations between particular model levels and source code, the method is targeted to achieve the applicability in the area of the iterative, incremental and model driven development.

Keywords: Design patterns, semantics, transformations, concretization, specialization, variability, models of design patterns, model driven development

Mathematics Subject Classification 2010: 68N19

1 INTRODUCTION

The concept of patterns was first introduced in the work of Alexander [1] dealing with urban solutions, but soon the patterns were defined and used in software engineering as well. The idea of applying verified pattern solutions to common recurring problems in the software design attracted considerable attention very quickly, since

the quality of software systems depends greatly on the design solutions chosen by developers.

Patterns have been applied in various phases of the software development lifecycle. Patterns were discovered and defined in software analysis, design, integration, testing and other areas [21]. Currently, design patterns represent an important tool for developers in the process of software design construction, and provide especially effective ways to improve the quality of software systems. Software development teams are capable to produce better software effectively thanks to patterns application. Consequently, the suitable tool based support of design patterns has great significance. As Berkane et al. [2] noted, using variability and design patterns can be very productive in improving adaptability of the software systems under development.

Section 2 introduces several related approaches to the design pattern support and Section 3 concludes with the open problems in the problem area. Section 4 explains the presented approach of design pattern support also with its realization and implementation. Next, Section 5 contains the evaluation of the method in form of case studies. The paper is completed by summarization and proposals for the future work.

2 RELATED WORKS

There exist several related approaches which introduce their own tool-based support of design patterns.

Mapelsden et al. [14] introduce an approach to design pattern application based on the Design Pattern Modelling Language. The authors describe the language, which is a notation for the specification of solutions of design patterns and their instantiation into UML models. Design pattern instances are regarded as part of the object model, providing another construct that can be used in the description of a program. Once all design pattern instance elements are linked to one or more UML design elements, consistency checks are made. A deficiency of this approach is that the developer needs to model all pattern participants manually and then link these parts to the pattern model.

Another method was introduced by Ó Cinnéide et al. [17]. They present a method for the creation of behavior-preserving design pattern transformations and apply this method to GoF design patterns. The method involves a refactoring process which provides descriptions of transformations to modify the spots for pattern instance placement (so called precursors). Placement is achieved by the application of so called 'micropatterns' to the final pattern instances. While Ó Cinnéide's approach is supposed to guide the developers pattern placement in the phase of refactoring (based on source code analysis), Briand et al. [3] try to identify the spots for pattern instances in the design phase (based on UML model analysis). They provide a semi-automatic suggestion mechanism based on a decision tree combining an evaluation of the automatic detection rules with user queries.

All the former approaches focus on the creation of pattern instances. The ones presented by Dong et al. [8, 7] presume the presence of pattern instances in the model. They provide support for evolution of the existing pattern instances resulting from application changes. In the former [8], the implementation employs QVT based model transformations, and in the latter [7] the same is achieved by XSLT transformations over the model stored as XMI. However, both work with a single configuration pattern template allowing only changes in the presence of hot spots participants. Other possible variations are omitted.

Debnath et al. [5] propose a level architecture of UML profiles for design patterns. Authors introduce a profile for patterns and analyze the advantages of using profiles to define, document, and visualize the design. Authors provide a guide to the creation of UML Profiles, but they give no concrete way of providing support in any tool. Dong et al. [6] discuss some of the relevant aspects of the UML profile. The paper presents an approach to the creation of UML profiles for design patterns. The approach allows an explicit representation of design patterns in software designs and introduces a notation for names of stereotypes and tagged values: Type<name:String [instance:integer], role:String>; for example: PatternClass<Observer [1], ConcreteObserver>. The introduced notation is useful because it visualizes individual instances of design patterns, but the Type part of the notation is redundant because the stereotype definition itself already carries this information.

Meffert [15] introduces an approach assisting developers in selection of the correct design pattern for a given context. The approach introduces the annotations to the source code in order to express an intention of the given source code fragment. Meffert also proposes the description of the intention for some design patterns. The suitable pattern is recommended to a developer on the basis of comparison of the annotated source code intention with the intention defined for the design pattern's parts.

Sabo et al. [19] present a method of preserving the correct form of applied design patterns during the process of software system evolution. The method aims to explicit indication of the pattern participants in the source code by annotations. The authors also propose a mechanism determining whether the applied pattern instances are still valid or have been broken due to a meantime code modification.

Kirasić et al. [12] present an ontology-based architecture for pattern recognition. The authors integrate the knowledge representation ground and static code analysis for pattern recognition. Goa at al. [10] created an on-line repository for embedded software to facilitate component management and retrieval using ontology-based approach. This approach can be effectively used for component-based development.

Another method of the patterns recovery based on code annotations and regular expressions has been introduced by Rasool et al. [18]. The authors extend the list of annotations defined in [15] in order to detect the similarity of different annotations used in multiple patterns. Authors' intention is to use the annotations for the static analysis of the source code and subsequent recognition of structural design patterns.

Fülleborn et al. [9] present an approach of the documentation of the particular source code or UML models that have design deficiency, in order to document the

problems in their context that the chosen design pattern solves. Documenting is done by adding non-functional requirements in form of annotations. Next, the authors formally document also the solved problems so that they can be compared to the situation before the chosen design pattern was applied. By the way of comparison, the transformation between the situation before and after the application of the design pattern is made explicitly in order to derive the reusable cross-domain representation of the situation. One of current challenges in the area of Model-Driven Architecture (MDA) is transformation of Computation-Independent Models (CIM) to Platform-Independent Models (PIM), which usually requires a thorough understanding of domain, expertise and creativity, which enhances the difficulty of this transition [11, 4]. In our work, we have focused on PIM to (Platform-Specific Model) PSM transformation.

3 OPEN PROBLEMS

Most of the current approaches focused on the pattern support only at one level of abstraction and they do not provide any mechanism for preserving the pattern support also on other abstraction levels. For example, many of the approaches focus mainly at the design level (i.e. model), but by the transition to the source code level the pattern instances become almost invisible in the huge amount of source code lines without any further support. As a consequence, the evolution of the pattern instances is very difficult without any tool-based support, because a developer does not have a good view of all the participants of pattern instances in the source code. Moreover, due to the inability to identify the individual participants of pattern instances in the source code, they may be modified in an incorrect way during the system evolution and maintenance, and this may lead to a breakdown of the pattern and the loss of the benefits gained by its application in the software system.

Since the patterns provide abstracted and generalized solutions to recurring problems, their application to solve a specific problem requires to concretize and to specialize the solution described by the pattern [16].

Specialization process of a design pattern lies typically in its integrating into the specific context of the problem. This knowledge is mainly available to developers and domain experts involved in the design process, because it requires very specialized and detailed understanding of the domain context and the specific application. This is why this process is difficult to automate. Despite this, it is possible to make specialization of a pattern much easier by providing an appropriate mechanism for application of design patterns.

Goal of the concretization of a design pattern is to recast its abstract form into a concrete realization with all its parts, methods, attributes and associations, but only within the scope of the pattern instance and its participants, but not the rest of the application model. The more parts the structure of the pattern instance contains, the more concrete it becomes. The most concrete level of a design pattern

instance is the source code, because at this level of abstraction the pattern instance contains all parts from its structure. Majority of activities in the concretization process depends on stable and fixed definition of the design pattern structure so that these activities are fairly routine. This is a good starting point for automating the process.

CASE or other modeling tools and approaches provide today some kind of support for design pattern instantiation, but it is often based on simple copying of pattern template into the model with minimal possibilities for modification and with minimal support for instance integration into the context – application model.

Similarly the approaches that focus on creation of pattern instances are typically based on strict forward participant generation – participants in all roles are created according to a single template. Likewise as the support of design patterns available in the traditional CASE or other modeling tools is usually based on UML templates of each design pattern. So they are simply copied into the model with a minimal possibility for modification and integration in the rest of model when pattern instance is created. However, patterns describe not only the main solution, but also many alternative solutions and variations. But a developer is not allowed to choose an appropriate variant or concrete structure of the design pattern. Only one generic form is offered to the developer for use. Any other adjustments need to be performed manually without any tool based support. Moreover, the instances of patterns created by a tool is typically without any connection to the rest of the application model. So the instance of a pattern is not integrated into the application model, into its context. Associations are missing and the names of pattern participants are general, and so on. All these activities of instance specialization have to be done by the developer manually. Even in the approach presented in [14], the developer needs to model all pattern participants manually, and then link these parts to the pattern model.

Our intention is to automate these activities. Our vision is that the developer simply suggests and specifies a pattern instance occurrence directly in the context via semantic marking of context elements, and the rest of the pattern structure is then automatically generated by subsequent transformations of models into the appropriate form.

4 DESIGN PATTERN SUPPORT BASED ON PRINCIPLES OF MODEL DRIVEN DEVELOPMENT

The main ideas of the approach are presented in Subsection 4.1. The solution of design pattern semantics definition and expression in the model and source code are described in Subsection 4.2. Subsection 4.3 presents the main support of design patterns and Subsection 4.4 explains the continual support of design patterns in the source code.

4.1 Main Ideas

The abstraction, semantics and model transformations represent the key principles of Model Driven Development and Model Driven Architecture. Thanks to these principles, the automation of many aspects of the system development can be achieved. The semantics applied in the models enable the possibility to understand the model and its elements, and also to recognize which elements play which roles in the model. Consequently, on the basis of understanding the models and their elements, it is possible to construct the transformations which transform the models to a lower level of abstraction. These principles represent the basis of the elaborated method of design pattern support.

Patterns are often being described as a collection of cooperating roles. Our approach is based also on the idea that the pattern roles can be divided into roles dealing with the domain of the created software system and roles performing the pattern's infrastructure. The domain roles can be considered as the "hot spots" when they can be modified, added or deleted according to the requirements of the particular software environment. The roles performing the pattern infrastructure are not changing too much between the pattern instances. Their purpose is to glue the domain roles together to be able to perform desired common functionality. The examples of domain dependent roles are presented in Table 1.

Employment of the patterns into the project allows the developer to think about higher level of abstraction. When he/she decides to employ the pattern, the first thing he/she needs to take care of is how it will be connected to his/her project, how the solution will be integrated to the rest of his/her model or code. At this moment he/she does not focus on the entire inner structure of the patterns, at this moment it is irrelevant. The way how he/she integrates the pattern to the project is in specification of the domain roles. Their participants can be existing parts of the project or new parts of the project created for this situation. Once the domain roles are specified, the specification of the infrastructure roles takes place. This is quite a routine when the developer subsequently adds participants of the infrastructure roles according the sample instance from the pattern catalogue.

When we look closer on such instantiation process from the perspective of its division into two more or less independent processes specialization and concretization (described in Section 3 "Open Problems" [16]), we can see that the user does the specialization process when he/she is specifying the domain roles. When he/she is supplementing pattern instance with the infrastructure roles he/she only finishes the concretization process. In our approach we do not want to replace the developer in the specialization process, but we want to relieve him/her of the necessity to instantiate the infrastructure roles meanwhile the concretization process. We want the developer to make a suggestion by the application of semantics as to where and which design pattern he/she wishes to be applied in the model and to specify the domain dependent roles. Then he/she can also specify which variant of the pattern to employ, and in what way he/she wants it to generate. Subsequently the rest of the pattern instance structure will be automatically generated by model

Pattern	Domain Dependent Roles	Description
Composite	Leaf and its Operations	Leafs and their operations provide all domain dependent functionality. Everything else is just infrastructure allowing the hierarchical access to the leaf instances.
Flyweight	Concrete Flyweight	Concrete Flyweight provides all domain dependent functionality. The rest is infrastructure for storing instances in memory providing access to them.
Proxy	Real Subject, Proxy	The domain dependent is the Real Subject (which often exists before Proxy pattern application) and functionality of Proxy participants that provide access to the Real Subject.

Table 1. Examples of domain dependent roles of patterns

transformations to lower levels of abstraction according to the instance suggestion and specification.

In case the transformations are driven by an appropriate model of design pattern, and both the model of an application and the model of the pattern contain information on semantics, the transformation is capable of comparing these models and to create mappings between them. So in this way the transformation can recognize participants of design patterns that are present in the application model already, and which are not. As a consequence, the transformation is able to generate missing participants in a desired form obtained from the pattern model.

Moreover, we try to support the design patterns at more levels of abstraction in accord with ideas of the MDA development process. The elaborated approach provides the support of design patterns at three levels of abstraction:

1. suggested and specified platform independent instances of design patterns in the model (PIM),
2. more concrete and platform specific instances of design patterns in the model (PSM),
3. concrete and application specific instances of design patterns in the source code.

4.2 Semantics of Design Patterns

In order to achieve the specified goals, it is necessary to provide an appropriate mechanism of pattern semantics in the application model and source code. It is important to support insertion of semantics directly onto the elements of the model or source code, because such approach supports the specialization of pattern instances and makes the creation of the instances specification effortless. Thanks to

the semantics, these model transformations are able to understand the model of the application and recognize its parts.

4.2.1 Semantics of Design Patterns at Model Level

We choose the semantic extension of UML in a form of UML profile as a standard extension of UML, since one of our goals is to remain compliant with the majority of other UML tools. UML profiles provide a standard way to extend the UML semantics in the form of definitions of stereotypes, tagged values – meta-attributes of stereotypes, enumeration and constraints. All these elements can be applied directly onto specific model elements such as Classes, Attributes, and Operations [13]. In this way it is possible to specify participants of design patterns and relations between them directly in the context – on the elements of the application model. The snippet of UML profile for Observer pattern is shown in Figure 1. Authored UML profile for design patterns provides semantics to various pattern instances adjustments, suggestions and specifications. However, it is not mandatory to apply all the semantic elements (stereotypes). The developer applies and specifies only what he needs to express. Because of the default values of meta-attributes of stereotypes, the transformation always has enough information for default behavior. Inconsistent specifications of pattern instances can be handled by OCL constraints which are part of UML profile as well.

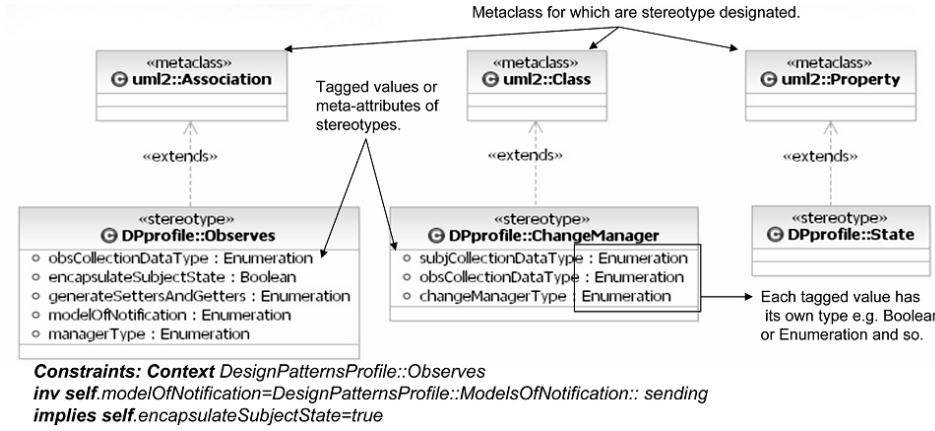

Figure 1. The snippet of UML profile with some elements for Observer pattern

4.2.2 Semantics of Design Patterns at Source Code Level

Source code annotations work as metadata information for different artifacts and fragments of the source code. This information can be processed by various tools (compilers, etc.). Thanks to the source code annotations, the semantics and visibility

of patterns can be preserved and propagated from model also into the source code. We propose the following definition of annotation for design patterns (see Figure 2).

```
public @interface DesignPattern{
    PatterNames patterName();
    String instanceAlias();
    RoleNames roleName();
    String variant() default "DEFAULT";

    enum PatterNames{ Observer; //...
    }
    enum RoleNames{Subject, attach, detach, notifyObservers,
        update, observers, Observer, ConcreteObserver; //...
    }
```

Figure 2. Definition of the source code annotation for design patterns

The attribute patternName of the annotation expresses the name of the pattern, e.g. Observer, Mediator, Command, etc. Because one pattern (for example Observer) may have more different instances applied, the pattern instance "alias" is necessary for the recognition among these instances. The attribute roleName expresses the name of the pattern participant, e.g. Subject, ConcreteSubject, Attach, etc. Some participants of the pattern instances may have more possible variants and therefore the attribute variant is also necessary.

The presented proposal is intended for Java platform, but it can be simply adjusted also for other platforms, even if they do not support source code annotations. In such case the annotations may be enclosed in comments. However, because Java does not support the annotation of one code unit (i.e. method, class, etc.) by more than one annotation with the identical name, this approach is limited in the case that one fragment – unit of the code – represents more roles in more patterns (for example, in the case of pattern composition). This problem can be resolved by enclosing the next DesignPattern annotations in comments as well.

4.3 Design Pattern Support

In the first step the developer suggests pattern instance occurrence by the insertion of semantics, i.e. application of stereotypes into the model. In the second optional step, the developer specifies a desired variant or configuration of instance by setting tagged values of inserted stereotypes. Then he runs the transformation to a lower level of abstraction. The transformation generates the rest of the pattern, and also marks the participants of the pattern. From the second step the process can be repeated at lower level of the abstraction. The only difference is that at the lower level of abstraction (PSM) in the second step, more implementation dependent choices (e.g. data types) are offered which the developer was not asked previously at the higher level (PIM). The overall illustration of design patterns support process is illustrated in Figure 3.

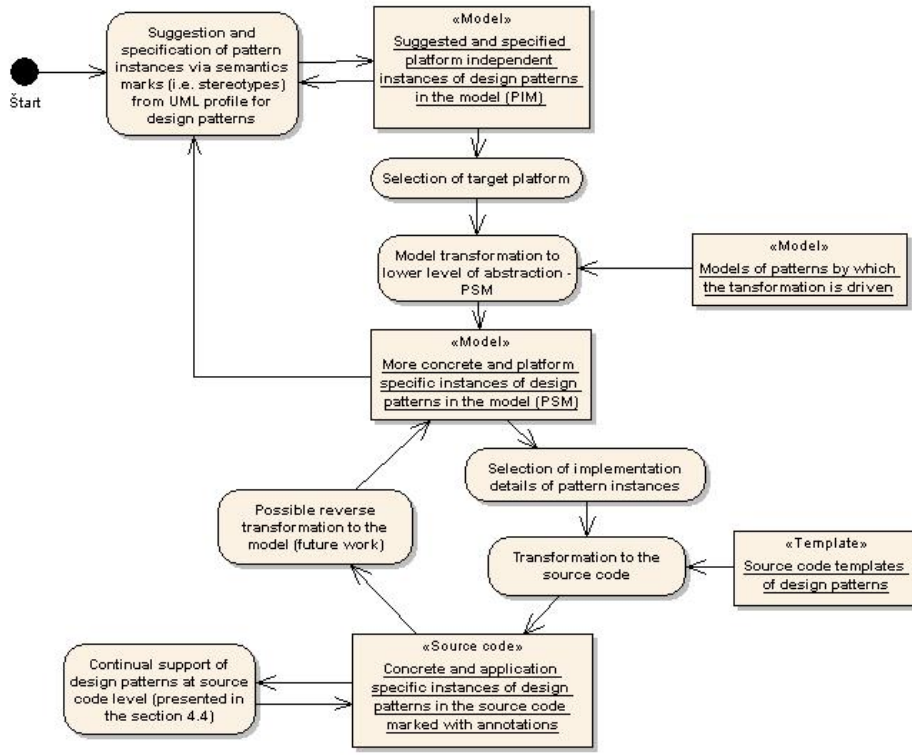

Figure 3. The overall illustration of design pattern support process

The suggestion and the specification of pattern instance are realized by applying information on the semantics into the models provided by semantical extension of UML. For example, Figure 4 shows a suggestion of the Observer pattern instance via applying one stereotype $<< Observes >>$ to a desired element, in this case, an association. From this information the transformation can recognize that the source element of the association represents a Concrete Observer and the destination element is a Concrete Subject. Consequently, on the basis of this information and the available pattern model and semantics, the transformation can recognize other missing pattern participants which must be added into the model. The transformation also needs information how to generate the rest of the pattern instance, e.g. variant of pattern, desired adjustments of the pattern instance and so on. The next step is the specification of pattern instance. This goal is achieved by setting up values of meta-attributes of stereotype (see Figure 4). In our approach this step is not mandatory because default values of meta-attributes of the stereotype are set and are available. Consequently, the application of a desired pattern can consist only one suggestion mark – the stereotype – that can be

applied onto the specified model element, when the developer wants the default pattern variant. Any other activities will be completed by a tool via model transformations. In this phase, developers do not have to warry about the concrete details of the pattern structure and can comfortably work with pattern instances at a higher level of abstraction. Application of the desired pattern is realized on elements of the system model or context, and thus the specialization process is supported.

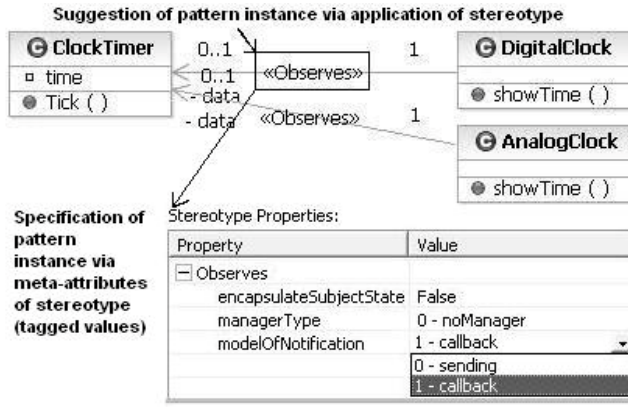

Figure 4. An example of the suggestion and specification of the Observer pattern instance into the model

The concretization process is realized and automated by model transformations to lower levels of abstraction until the source code level is reached. One of the possible results of the transformation of the model from Figure 4 is shown in Figure 5. As can be seen the transformation generates the rest of pattern structure in a desired form in accord with pattern suggestion and specification from the Figure 4. The pattern instance becomes more concrete, so the form of the instance now represents its lower abstraction level. Thanks to the realization of the pattern instance by placing the suggestion and specification directly into the context of elements in the application model, the transformation is also able to integrate the generated participants with participants already present in the model. As a result, the pattern instance is in the application specific form.

It is important that the transformation is realized and launched with a choice of target platform, because at this point the first differences may occur in the structure of pattern instances depending on a target platform. The choice of a target platform also determines the set of possible choices of data types before the subsequent transformation to the source code level. As one can see in Figure 5, the transformation also adds explicit marks (stereotypes) to all identified and generated pattern participants. The addition of marks and also the whole transformation is performed on the basis of the pattern model. As a consequence, the instances are

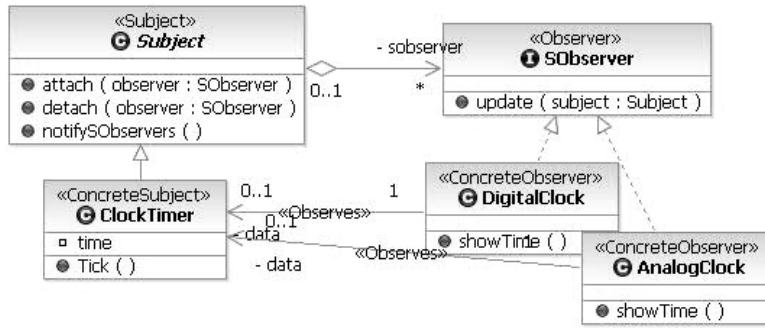

Figure 5. The result of transformation to Java target platform of the model from Figure 4 in accord with the instance suggestion and specification

clearly visible in the models, and the developer can repeat the instantiation process at lower level (PSM) directly from the optional second step, i.e. specifying the instance and choosing a more detailed adjustments of pattern instance (e.g. concrete data types). Again, the default values of the stereotype meta-attributes are set, so the developer can run the transformation to source code directly.

So the models with concretized instances of patterns are transformed into the source code in the next step. In order to propagate the visibility of the applied patterns from the model into the source code we have used proposed annotations (see Figure 2). In Figure 6 the source code snippet of Subject generated from the model in Figure 5 is illustrated. Each generated pattern participant is annotated with the proposed definition of annotation. The transformation of the model into the source code is made in the form of source code templates which also generates the pattern participants with correct annotations. For classes marked with a stereotype, the template with the same name as the stereotype name is used. For example, for the model classes marked with the stereotype $<< Subject >>$, the template with the name `subject.javajet` is used, etc.

Consequently, in this approach we propagate and expand two applied stereotypes from higher level of abstraction (i.e. $<< Observes >>$ from Figure 4) onto lots of annotations in the source code (e.g. Figure 6 – however, it is only a small snippet from one class). So this way, the huge manual annotation of pattern participants in a large source code is not required and it is reduced to a little manual suggestion via stereotypes at the highest level of abstraction (e.g. Figure 4).

Moreover, two separate groups of classes are generated by the initial transformation to source code. The first one is the base group which is always overwritten by subsequent source code generation (see Figure 6 – `SubjectBaseclass`). The second one is the development group which is generated only by initial transformation. The developer can write and add a specific implementation here without the threat of overwriting.

```
package pk.annotations;
import java.util.ArrayList;
import pk.annotations.DesignPattern.PatterNames;
import pk.annotations.DesignPattern.RoleNames;


@DesignPattern( patterName = PatterNames.Observer,
        instanceAlias = "obs1",
        roleName = RoleNames.Subject,
        variant = "encapsulateSubjectState = false; managerType = noManager")
public class SubjectBase  {

    @DesignPattern( patterName = PatterNames.Observer,
            instanceAlias = "obs1",
            roleName = RoleNames.observers)
    ArrayList<SObserver> observers = new ArrayList<SObserver>();
    //...

    @DesignPattern( patterName = PatterNames.Observer,
                    instanceAlias = "obs1",
                    roleName = RoleNames.attach)
    public void attach(SObserver observer){
        observers.add(observer);
    }
    //...

    @DesignPattern( patterName = PatterNames.Observer,
            instanceAlias = "obs1",
            roleName = RoleNames.notifyObservers,
            variant = "modelOfNotification = callback")
    public void notifySObservers(){
        for(SObserver observer: observers)
            observer.update(this);
    }
    //...
}
```

Figure 6. The source code snippet of Subject generated from the model in Figure 5

4.3.1 Realization of Transformations

The model transformations are driven by properly specified and marked models of design patterns. These prepared models cover all supported pattern variants and possible modifications. Each element of these models is marked. There are two types of marks in pattern models. The first type of marks expresses the role of the element in the scope of the pattern. On the basis of this type of marks the tool is capable of creating mappings between models. The second type of marks expresses an association of the element with a variant of the pattern. On the basis of this type of marks the tool is capable of deciding which element should be generated into the model, which way and in what form. For the second type of marks the following notation is defined:

$$[\sim]\ \mathit{StereotypeName} :: \mathit{Meta\text{-}attributeName} :: \mathit{value};$$

An element from the pattern model is generated into the model only if the specified meta-attribute of the specified stereotype has the specified value. These marks can be joined via *";"*, while the symbol *"∼"* expresses negation. If an element has no

mark, it is always generated into the model. A sample section of the pattern model of the Observer pattern is exposed in Figure 7.

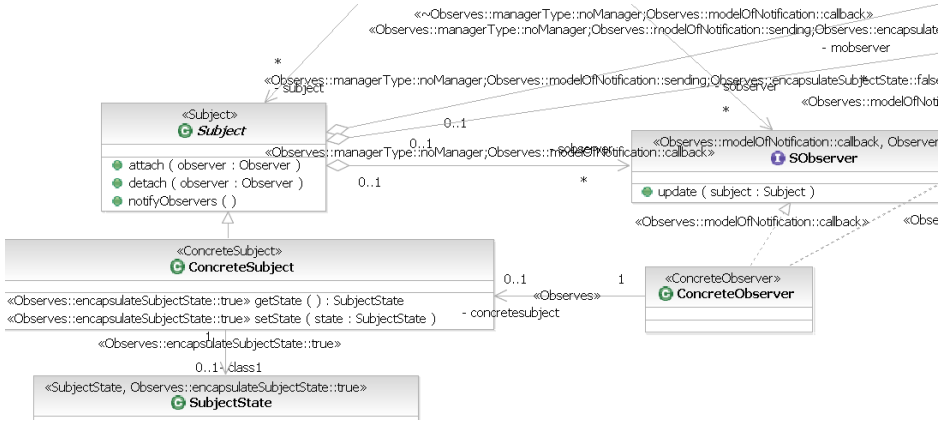

Figure 7. Sample section of Observer pattern model by which the model transformations are driven.

The first action performed by the tool after the transformation started is the comparison of the first type marks in the pattern model to the marks in the application model. Based on the first type marks comparison the tool is capable to do mapping between the marked models, and consequently to recognize which parts of the structure of the design pattern instance are in the model of the developing application and which are not. For example, in Figure 4 we have shown the application of the Observer pattern by applying two stereotypes $<< Observes >>$ on the directed association. The tool can from this way marked association recognize that the pattern roles Concrete Observer and Concrete Subject of this two Observer pattern instances are already present in the model, and also which elements (in this case classes) in the application model represent these pattern roles.

Decisions which variant of the pattern and which elements from the pattern model need to be generated into the application model are based on the comparison of the second type marks in the pattern model with the values of meta-attributes of stereotypes. These values are set up by the developer in the second step – the specification of the pattern instance (for example see Figure 4). After decision-making and selection of the desired pattern form, the final transformation is performed from the pattern suggestion level to the lower level of abstraction. The results of the transformation are correctly specialized and concrete instances of the patterns are in the desired form (for example see Figure 5).

Driving the model transformations by pattern models allows us to adjust results of transformations by modifying the pattern models which drive the transformations. Marks in the models ensure that the tool is always capable of creating correct mappings between the model of application and the model which drives the transfor-

mation, and consequently decide which element should be generated into the model and in what form. This way it is possible to model any custom structure and achieve support for its application into the model.

The transformation to source code is realized on the basis of the source code templates. Each pattern participant has own source code template. The transformation takes source code template with name identical to the stereotype name of the participant and it generates template's content into a specified destination. For model elements without any stereotype the common code template is used which generates only signatures of the class, fields and methods with empty body.

Implementation

The presented support and transformations were implemented and verified in the form of the IBM Rational Software Modeler transformation plug-in. The first type of transformation of the model of the highest level of abstraction to the model of the lower level of abstraction was implemented by M2M, UML2 and EMF frameworks. These frameworks are subprojects of the top-level Eclipse Modeling Project and they provide ideal infrastructure for model-to-model transformations.

The second type of transformation of the model with a lower level of abstraction (PSM) to source code was implemented by frameworks JET, UML2 and EMF. JET is also part of the Eclipse Modeling Project in M2T (Model to Text) area. It provides infrastructure for source code generation based on code templates.

4.4 Continual Support of Design Patterns at Source Code Level

The annotations of patterns generated into the source code by designed transformation to the source code (see Figure 6 in Section 4.3.) highlight the visibility of pattern instances and therefore make identification of pattern participants in the source code easier. In consequence, the support of the pattern detection, instantiation and evolution in the source code can be achieved in a very suitable form of a source code context assistant. Thanks to annotations, the support mechanism will be able to identify the pattern participants already implemented, and subsequently it will be able to offer an option to generate any missing pattern participant or to perform possible pattern evolution in the given context, etc. This idea brings significant improvement of the pattern support at the source code level.

4.4.1 Support of Design Pattern Instantiation and Evolution

The support of the pattern instantiation and evolution is realized in form of the source code context assistant with the consequent source code generation. The result of the source code generation depends on the expression of by developer typed annotation and its location in the source code. The method is described in the following steps.

1. In the first step, the developer begins with writing of the proposed pattern annotation (see Figure 2 in the Section 4.2.2) in the desired location in the code. When the developer writes `@DesignPattern( patternName =`, the context assistant offers the set of names of supported patterns. The developer, for example, chooses `PatternNames.Observer`.
2. Next the developer continues with writing of the annotation and writes `instanceAlias`. So the annotation looks as follows: `@DesignPattern( patternName = PatternNames.Observer, instanceAlias =`. Now the context assistant searches all the existing instances of the pattern with the given name i.e. `PatternNames.Observer` and it offers the developer the set of aliases of all existing instances of Observer pattern in the project. Because of the suitable annotation structure this search is very straightforward.

Consequently, the developer chooses an `instanceAlias` from the offered set or writes a new, unique alias. When the developer writes a new, unique instance alias, the support mechanism deduces that the developer desires a creation of a new pattern instance. Otherwise, when the developer chooses one of the offered existing instance aliases, the support mechanism deduces that the developer desires evolution of the pattern instance identified by the chosen instance alias and the pattern name. According to the developer's choice pattern instantiation or evolution follows.

Design Pattern Instantiation

When in the second step the developer wrote a new, unique instance alias, the instantiation of the pattern with the typed name is performed (in our case instantiation of Observer). The method continues with the following steps.

3. The support mechanism loads feature model of the pattern. It selects all mandatory features at the first level (i.e. classes) and generates them into the source code.
4. If one of the mandatory features has more possible variants, the developer is asked for selection of its variant via dialogue during the instance generation.

Illustration of the feature model of Observer pattern is shown in Figure 8.

The first mandatory class is generated at the position of the entered annotation in the current file, therefore in case of the pattern instantiation the developer should write the annotation in a new empty file. Other mandatory classes are generated into new automatically created empty files in the current package of the project. Of course, an element is always generated with all its mandatory sub-elements.

Design Pattern Evolution

When in the second step the developer selects alias from offered set of all existing instance aliases of the pattern with the typed name (see step 2), the support

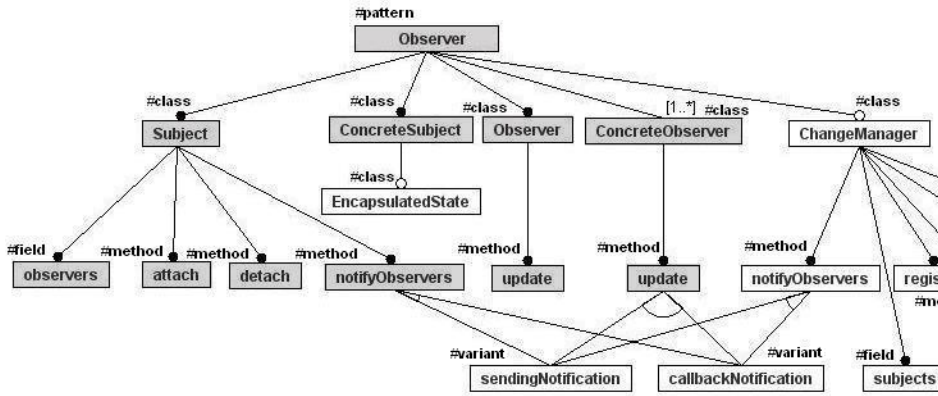

Figure 8. Illustration of feature model of Observer. Mandatory features are filled with gray color.

mechanism deduces that the developer wants to perform the evolution of the pattern instance with the selected instance alias. The support continues with following steps.

3. The support mechanism creates a feature model configuration of the pattern instance identified by the selected alias. Thanks to the annotations, the recognition of the pattern instance participants present in the source code is quite easy.
4. The support mechanism loads the feature model of the pattern.
5. The created feature model configuration of the pattern instance is compared with the loaded feature model of the pattern. In consequence, the options of possible evolution of the pattern instance are detected (see Figure 9).
6. The support mechanism offers the detected set of possible options of instance evolution in the form of the context assistant (see Figure 10). So the developer may choose the desired pattern instance evolution.

It is important to remark that only the roots of possible instance evolution subtrees are offered to the developer, because generation of child elements (e.g. methods) has no sense as long as the parent element (e.g. class) does not exist in the source code. The selected element with all its mandatory sub-elements is generated at the position of the entered annotation in the current file. So the method supposes at least basic knowledge of patterns. If an element has more possible variants within the scope of given instance, the developer is asked to select one of the variants via the dialogue during the element generation. Within the scope of the pattern evolution also the detection of missing mandatory features is supported (for example see Figure 9, the update method of Observer instance is missing). This way the basic check of the pattern instance validity is achieved.

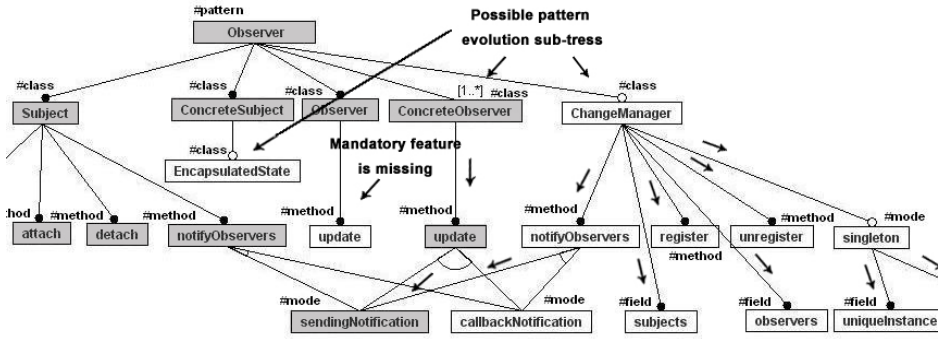

Figure 9. A comparison of the feature model configuration of an existing Observer instance with the feature model of Observer pattern (existing participants – features are filled with gray color). The possible options of pattern instance evolution are illustrated by arrows.

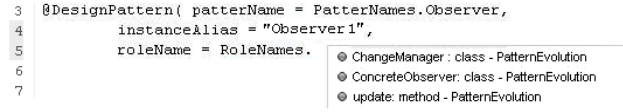

Figure 10. Example of detected set of possible options of instance evolution offered to the developer in the form of context assistant

4.4.2 Realization

Each element of the pattern feature model (except the elements marked as #pattern or #variant, see Figure 8 or 9) has its own code template attached. Each code template of an element includes subsequent templates of all related mandatory sub-elements of the element in accord with the feature model of the pattern. Therefore an element is always generated with all its mandatory sub-elements. For example, `Subject` template includes `observers, attach, detach` and `notifyObservers` templates. Example of `Subject` template is illustrated in Figure 11.

If an element has more possible variants, the template of such element contains the source code for all variants distinguished by annotations (for example, see Figure 12). The following notation has been introduced for the variant attribute of proposed annotations from the Section 4.2.2, Figure 2:

$$[\sim]\ Attribute\_name = value[;]$$

If the attribute value selected by the developer in GUI dialogue corresponds with the introduced notation, the variant of an element is generated from the template. Dependency on more than one value or attribute can be attached via *";"*, while the symbol *"∼"* expresses negation (it is based on the analogical principles as presented notation for marks by model transformations in Section 4.3.1.). So when the

```
<%@ jet class="Observer-SubjectTemplate" package="dp.anot.jet" startTag="<%" endTag="%>"
<% Map context = (Map)argument;
String instanceAlias = (String)context.get("instanceAlias");
String curr_package = (String)context.get("package");
//...
%>
package <%=curr_package%>;
import java.util.*;

@DesignPattern( patterName = PatterNames.Observer, instanceAlias =  <%=instanceAlias%>,
        roleName = RoleNames.Subject )
public class Subject<%=instanceAlias%> {

  /*included and generated subparticipants */
  <%= includeParticipant("observers") %>
  <%= includeParticipant("attach") %>
  //...
  <%= includeParticipant("notifyObservers") %>
}
```

Figure 11. Example of Subject template. The template includes subsequent templates of sub-elements of Subject in accord to the feature model of Observer pattern.

element – feature has more than one possible variant, the developer's selection is compared with annotations in the template and in consequence, the desired variant of element – feature is generated.

As it can be noticed, in Figure 11, the names of new generated classes, methods and fields are created as `roleName+InstanceAlias`. The developer may rename the elements later, of course. However, when a body of a method is generated in the scope of an instance evolution, the introduced name convention is not sufficient enough. The bodies of generated methods should be tied to an existing implementation of the instance and therefore the particular names of existing elements should be found out (for example, see `observerClassName` retrieving in Figure 12). Because of the annotations of existing pattern participants this is a straightforward task.

Moreover, the whole method is based on the following defined name conventions. The names of feature models are identical to the `PatternNames` used in the source code annotations and the feature names are identical to the `RoleNames` used in the source code annotations as well. The templates are named as follows: `PatternName-RoleNameTemplate`. As a consequence, the support mechanism is able to automatically deduce from the annotations typed by the developer in the source code which feature model and which templates should be loaded and generated. This way the flexibility of the method is achieved and improved, since the addition of a new feature model and new templates is sufficient enough to extend the support for a new pattern. An extension of `PatternNames` and `RoleNames` about the new pattern name and roles is also necessary.

Implementation

Implementation of the method is based on the Eclipse platform. The templates are implemented in JET framework. The JET framework is a part of Eclipse Modeling

```
<%@ jet class="Observer-notifyObserversTemplate" package="dp.anot.jet" startTag="<%" endT
<% Map context = (Map)argument;
String instanceAlias = (String)context.get("instanceAlias");
String observersClassName = "";
if(instanceEvolution)
 observersClassName = getObserverClassName(PatterNames.Observer,
         instanceAlias,RoleNames.Observer);
if(observersClassName.equals(""))
 observersClassName = "Observer"+instanceAlias;
//...
%>
@DesignPattern( patterName = PatterNames.Observer, instanceAlias =  <%=instanceAlias%>,
        roleName = RoleNames.notifyObservers, variant = "modelOfNotification = callback")
    public void notifyObservers<%=instanceAlias%>() {
        for (<%=observersClassName%> o : <$=observersCollectionRefName>)
      o.<%=updateMethodName%>(this);
    }

@DesignPattern( patterName = PatterNames.Observer, instanceAlias =  <%=instanceAlias%>,
        roleName = RoleNames.notifyObservers, variant = "modelOfNotification = sending")
    public void notifyObservers<%=instanceAlias%>() {
        for (<%=observersClassName%> o : <$=observersCollectionRefName>)
      o.<%=updateMethodName%>(this.get<%=subjectStateClassName>());
    }
```

Figure 12. Example of notifyObservers template which contains two different variants distinguished by annotations (notice difference of variant attributes of annotations)

Project in M2T (Model to Text) area and it provides very good infrastructure for the source code generation based on code templates.

The feature models of patterns are implemented as UML class diagrams analogically as it has been introduced in [20] (see the section Feature Modeling Profile for UML), but for the method purposes we rather use the class diagram instead of the component diagram.

As mentioned earlier, Java does not support the annotation of one code unit (i.e. method, class, etc.) by more than one annotation with the identical name and so the current implementation is limited in case when one fragment – unit of the code represents more roles in more patterns (for example, in case of pattern composition). This problem can be resolved by enclosing the next `DesignPattern` annotations in comments. Similarly, the implementation can be simply adjusted also for other platforms, even if they do not support source code annotations, because the annotations may be enclosed in comments as well.

5 EVALUATION

The following subsections contain the evaluation of the presented method in form of case studies.

5.1 Detailed Case Study of Observer Pattern Application

This section provides detailed illustration of the method and the tool usage and functionality in example based way on case study of observer pattern application.

Figure 13 shows an example of initial form of UML model before application of patterns.

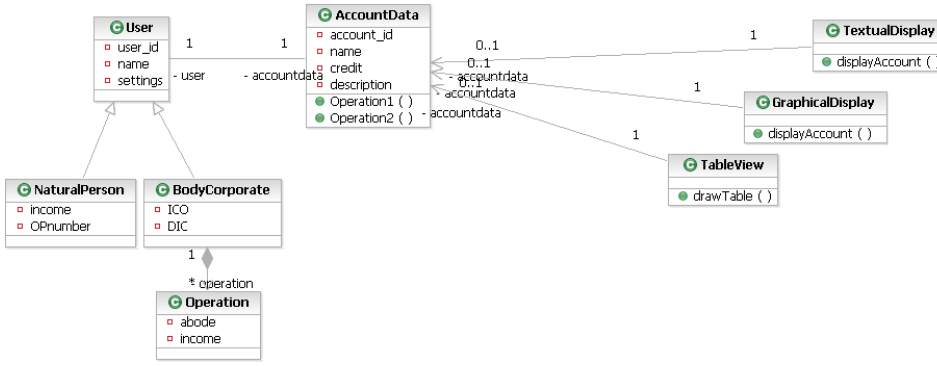

Figure 13. Example of starting UML model before the application of patterns

The model represents an example of starting point of the model and we want to apply, for example, Observer pattern into this model now. In order to apply a desired pattern (in our case e.g. Observer) we suggest the instance occurrences via particular semantics marks – stereotypes (in our case e.g. stereotype $<< Observes >>$). The suggestion of pattern instance occurrence via stereotype application is illustrated in Figure 14.

Notice that we perform the suggestion of pattern instance occurrence on existing model elements directly in the context and so in a consequence, the pattern instance will be integrated in the application model and context and thus any manual specialization of pattern instance is necessary. The resulting model after pattern instances suggestion is shown in Figure 15.

Now the tool knows what design pattern and where we want to apply it. Based on comparison of this model with the pattern model by which the transformations are driven the tool recognizes also that the association between classes `TextualDisplay` and `AccountData` corresponds with association between `ConcreteObserver` and `ConcreteSubject` from the pattern model. The recognition is made based on the first type of marks – stereotypes comparison in these models (see Figure 16) and in this way the tool makes mapping between these models.

Because the match of marks occurs on the association, the transformation recognizes that also source and destination elements of associations (in our case `ConcreteObserver` and `ConcreteSubject`) must be in the model of developing application already. In consequence, the transformation recognizes which elements of pattern model are in the model of application and which are not. Because the pattern model covers all pattern variants and possible modifications, the tool needs to

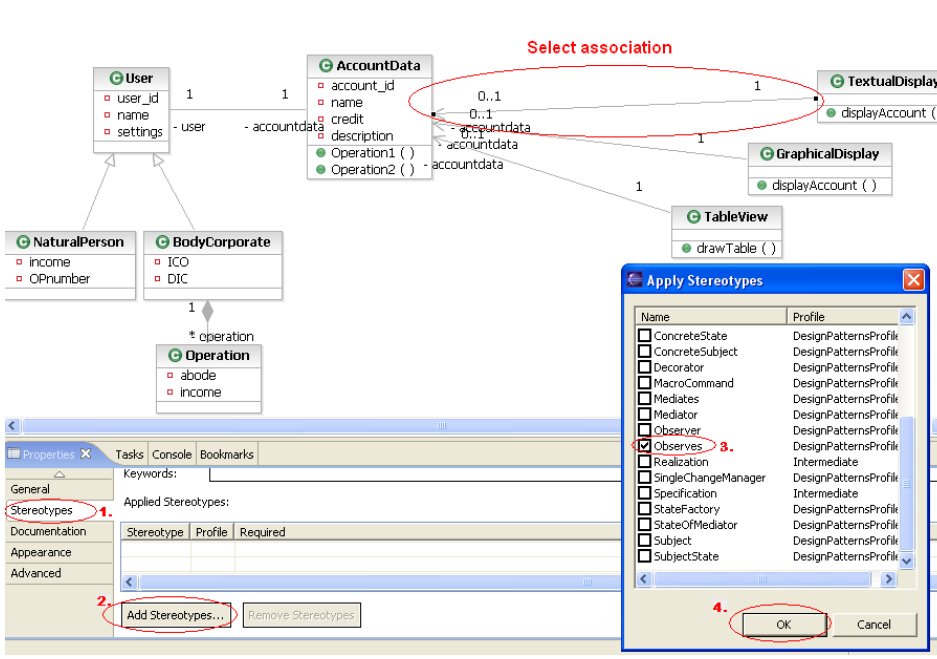

Figure 14. Application of stereotype $<<Observer>>$ on the selected association

know which variant or pattern modifications we want to generate. In other words, the tool needs to know which from all identified missing pattern elements from pattern model and in what way it should generate into the model of application. So we choose the variant and modifications of pattern instances via setting up values of particular stereotype meta-attributes in the next step of pattern instantiation (see

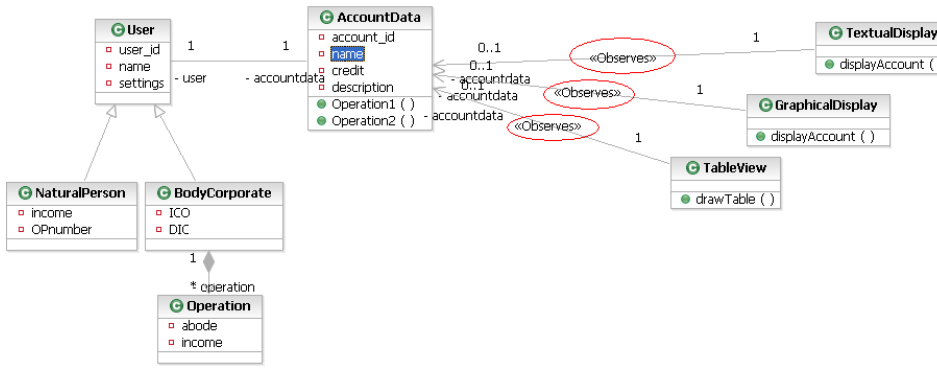

Figure 15. The resulting model after pattern instances suggestion

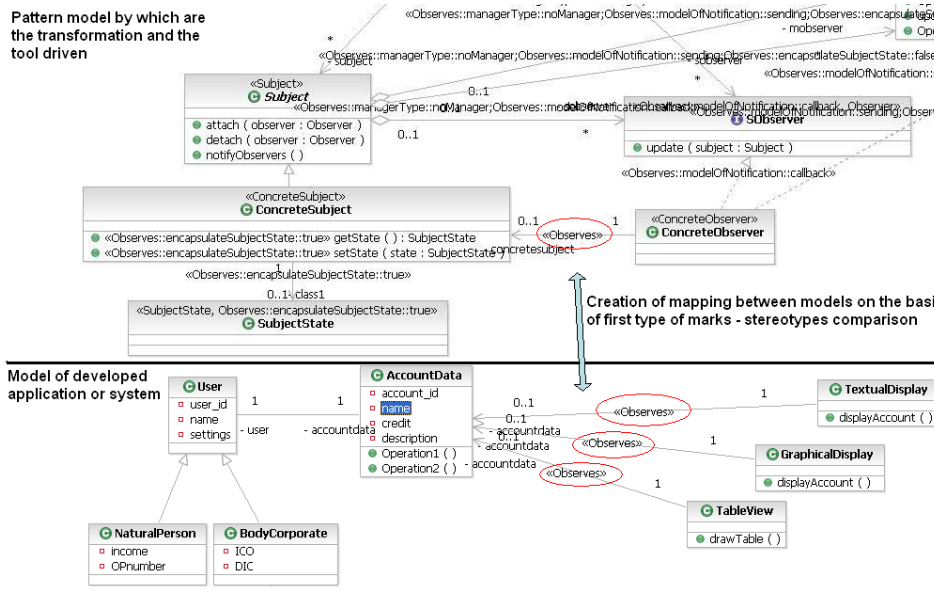

Figure 16. Creation of mapping between model of developing application and pattern model by which the transformation is driven

Figure 17). It is important to note that the meta-attributes of stereotypes have their default values set. Therefore this step is realized only if the developer wants to generate other than default variant of the pattern. The possible variants and adjustments of the pattern are defined in UML profile via enumerations or elements' primitive type specification such as Boolean, integer and so on.

We specify which variant or modification of the pattern we want and so we create specifications of suggested pattern instances via setting stereotypes meta-attributes values. Consequently, the pattern instances are suggested and specified. When the transformation is executing, the tool processes all identified missing pattern participants from pattern model and it checks the second type of marks – keywords on these missing elements. Section 4.3.1 describes how was it done in details. For the second type of mark the following notation is defined (note that these marks can be joined via *";"*, while the symbol *"∼"* expresses negation):

$$[\sim]\ ?\mathit{StereotypeName} :: \mathit{Meta\text{-}attributeName} :: \mathit{value};$$

A missing element from the pattern model is generated into the model, only if the specified meta-attribute of the specified stereotype has the specified value. Elements from pattern model, of which at least one second type mark does not match with the pattern instance specification, are ignored by the tool and so only elements with all positive matches of marks or without any mark are generated into the model. For example, when the `ConcreteSubject` element is identified as missing element, it

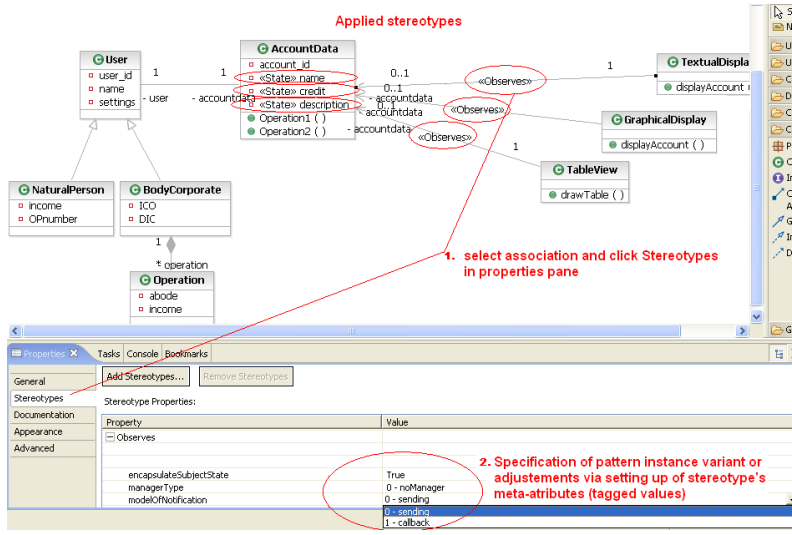

Figure 17. Specification of pattern instances via setting up of values of stereotype meta-attributes

is always generated into the application model, because it does not have any second type mark. On the other hand, the methods `getState` and `setState` are generated, only if the developer set value of meta-attribute `encapsulateSubjectState` of stereotype `Observes` on true, because these methods are marked with following second type mark (see Figure 18, *ConcreteSubject* class of Observer pattern model): *<<Observes :: encapsulateSubjectState :: true>>*.

Figure 18. *ConcreteSubject* element from Observer pattern model

When suggestions and specifications of pattern instances are completed, the transformation can be launched simply from context menu of the application model. The resulting model of transformation is shown in Figure 19.

The following sample specification of pattern instances has been set in the second step of pattern instantiation (see Figure 17).

1. *<<Observes>>* `AccountData TextualDispaly`:
   - `modelOfNotification = sending` – the interface of Observers which takes reference to the `SubjectState` class as notification parameter has been generated.

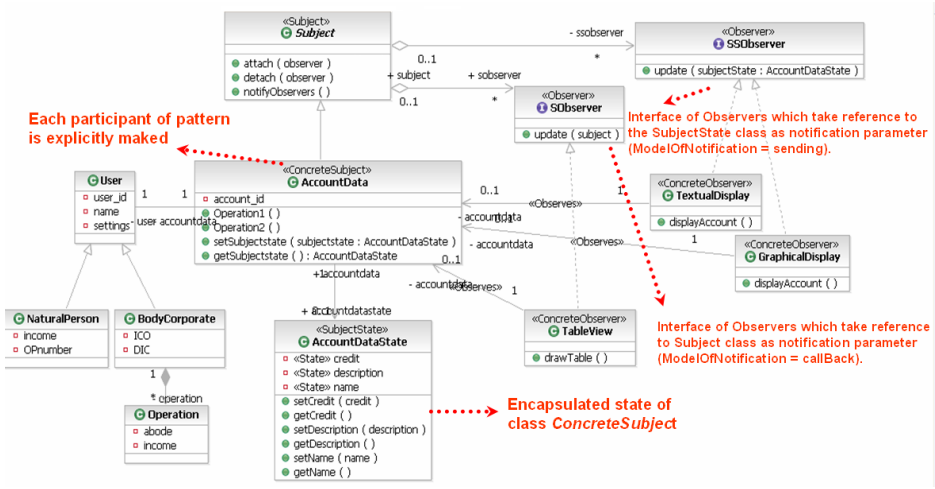

Figure 19. The resulting model of transformation of model from Figure 17

- `managerType = noManager` – no manager has been generated
- `encapsulateSubjectState = true` – the state of class `ConcreteSubject` has been encapsulated

2. *<<Observes>>* `AccountData GraphicsDisplay:`
   - the same as previous instance `AccountData TextualDisplay`.
3. *<<Observes>>* `AccountData TableView:`
   - `modelOfNotification = callBack` – the interface of Observers which takes reference to Subject class as notification parameter has been generated.
   - `managerType = noManager` – no manager has been generated
   - `encapsulateSubjectState` = false – this instance of Observer pattern does not use any encapsulated `SubjectState`, but the `Subject` reference instead.

The transformation explicitly marks also all identified and generated participants of pattern instances and in the consequence, it makes the participants clearly visible. Moreover, in the next step of instantiation the developer can repeat the previous instantiation process from the second step and can specify implementation details of pattern instances directly without necessity of further stereotype application (see Figure 20). This step is optional again, because the default implementations details are set and so the developer can launch the transformation to source code immediately. The snippet of resulting source code of transformation of model from Figure 20 to `Java` source code is shown in Figure 21.

The transformation to the source code generates two separate packages (`generated` and `developed`). The first is the base package which is always over-

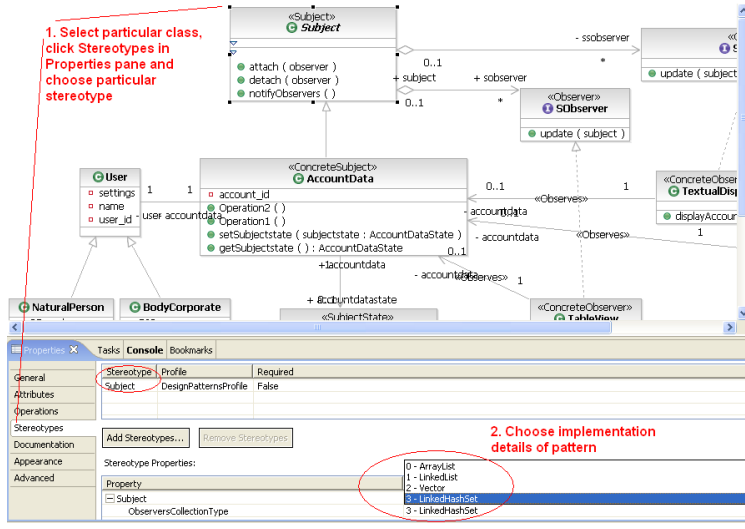

Figure 20. Specification of implementation details of pattern instances

written by subsequent source code generation. The second is the development package which is generated only by the initial transformation. Here the developer can write and add a specific implementation without the threat of overwriting. Further, two different methods of observers notification have been generated for each group of `Observers` in accord to their specification (in our case `TextualDisplay` and `GraphicsDisplay` as first group with `SSObserver` interface and `TableView` as second group with `SObserver` interface, see Figure 21). The transformation uses also the chosen data types (see Figure 20) in the source code generation and each participant of pattern instances is annotated with presented annotation for design patterns from Section 4.2.2.

After all, suggested and specified pattern instances from the highest level of abstraction have been transformed to the lowest level of the abstraction – the source code. The developer can utilize the created models and perform the next iteration of the development.

5.2 Experiment Results

The verification of the quantity and quality of the generated source code has been made by specifying an experimental task of creating an instance of Observer design pattern with four-watching objects (Concrete Observer) and an observed object (Concrete Subject). The time necessary to perform the task with the use of proposed method and tools, and without their use was measured. The amount of source code lines generated and written was measured too. Results are summarized in Table 2.

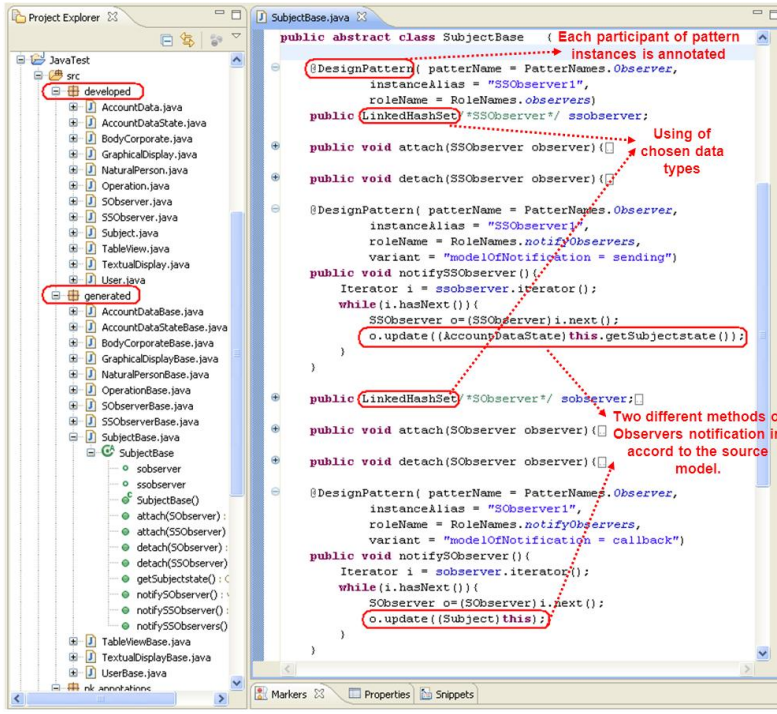

Figure 21. The snippet of resulting source code of transformation of model from Figure 20 to Java source code

Time needed using proposed tool	Time needed without using tool	Work speedup t2/t1	Code lines generated	Code lines typed in	Percentage of typed in code lines to whole source code
< 20 min	> 50 min	> 2.5	239	29	11 %

Table 2. Average results for the task of creating an instance of Observer design pattern with four-watching objects and an observed object

6 CONCLUSION AND FUTURE WORK

In this paper we presented the approach to design patterns instantiation support based on principles of model driven development. Semantics of patterns, which is introduced into the models via UML profile and into the source code via annotations, supports specialization process of patterns, because it is allowed to suggest and to specify the pattern instances participants directly on the context elements via application of specific semantic marks on them. Subsequent model transformations support and automate the concretization process of design patterns, because they

generate the rest of missing structure of suggested and specified pattern instances in a desired form and directly in context. Consequently, both of the processes (i.e. concretization and specialization process) of pattern instantiation are supported by the presented approach.

The transformations are driven by pattern instances suggestion and specification and by the pattern models as well. Transformations designed this way have several capabilities. First, they provide a possibility to choose an appropriate variant of the pattern by instance specification by setting up tagged values of the stereotypes. Second, they enable modeling of a custom pattern or structure by modification of the pattern model by which the transformation is driven, and in this way to achieve its generation into the model. The developer can model any custom structure, or even create a new one. As a result, the method is not oriented to the GoF design pattern support only, but it can also support other custom model structures which are often created in models mechanically.

The approach splits the details of a concrete design pattern instantiation into three levels of abstraction, and thus developers do not have to take care about concrete details of the pattern structure in the model of the highest abstraction level. Further, each generated pattern participant is annotated in accord to the described definition of source code annotations as result of the transformation to the source code.

The semantics of the patterns introduced into the source code by proposed annotations expands the visibility of pattern instances and it makes identifying of pattern participants in the source code easier. The clear visibility of pattern instances in the source code opens new opportunities to the support of various aspects of patterns as has been presented in Section 4.4. Furthermore, the introduced source code annotations enable also a correct reverse transformations of the source code to the model with the pattern detection and highlighting. Moreover, the available feature models of patterns also enable the possibility of live validation of pattern instances and detection of their defects in the source code.

Because manual annotation of the source code by developers is very lengthy and senseless, this approach provides useful way how to eliminate the manual annotation of the source code. The reduction of manual annotation is based on the idea of design information propagation and expansion from models of higher abstraction level into the source code. Although it does not deal with the problem of existing or legacy software systems, it provides very useful way how to propagate and expand design information and prevent the problem of pattern instances invisibility in source code toward the future. Besides, it does not have to be used only for patterns, but it can be simply adjusted also for other architectural or design decisions as well.

Nowadays, the approach does not give any guide on what patterns are suitable to apply. In our opinion, this guide is relatively hard to automate by the tool, because the knowledge of what patterns are suitable to apply requires really detailed understanding of the problem context and therefore, this knowledge is available especially to the developers or designers involved in the design process. But this is also a challenge to the future.

Acknowledgments

This publication is the partial result of the Research & Development Operational Programme for the project Research of methods for acquisition, analysis and personalized conveying of information and knowledge, ITMS 26240220039, co-funded by the ERDF and was partially supported by the grant No. VEGA 1/0752/14.

REFERENCES

[1] Alexander, Ch.—Ishikawa, S.—Silverstein, M.—Jacobson, M.—Fiksdahl-King, I.—Angel, S.: A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure). Oxford University Press, 1977.

[2] Berkane, M. L.—Seinturier, L.—Boufaida, M.: Using Variability Modelling and Design Patterns for Self-Adaptive System Engineering: Application to Smart-Home. International Journal of Web Engineering and Technology, Vol. 10, 2015, No. 1, pp. 65–93, doi: 10.1504/ijwet.2015.069359.

[3] Briand, L. C.—Labiche, Y.—Sauve, A.: Guiding the Application of Design Patterns Based on UML Models. 22nd IEEE International Conference on Software Maintenance (ICSM '06), September 2006, pp. 234–243, doi: 10.1109/icsm.2006.30.

[4] Cao, X.—Miao, H.—Chen, Y.: Transformation from Computation Independent Model to Platform Independent Model with Pattern. Journal of Shanghai University, Vol. 12, 2009, No. 6, pp. 515–523.

[5] Debnath, N. C.—Garis, A.—Riesco, D.—Montejano, G.: Defining Patterns Using UML Profiles. IEEE International Conference on Computer Systems and Applications, March 2006, pp. 1147–1150, doi: 10.1109/aiccsa.2006.205233.

[6] Dong, J.—Sheng, Y.: Visualizing Design Patterns with a UML Profile. Human Centric Computing Languages and Environments, IEEE, 2003, pp. 123–125, doi: 10.1109/hcc.2003.1260215.

[7] Dong, J.—Sheng, Y.—Zhang, K.: A Model Transformation Approach for Design Pattern Evolutions. 13th Annual IEEE Internet Symposium and Workshop on Engineering of Computer Based Systems, 2006, pp. 10–92, doi: 10.1109/ecbs.2006.10.

[8] Dong, J.—Sheng, Y.—Sun, Y.—Wong, W. E.: QVT Based Model Transformation for Design Pattern Evolutions. Proceedings of the 10th IASTED Internet Conference on Internet and Multimedia Systems and Applications, 2006, pp. 16–22.

[9] Fülleborn, A.—Meffert, K.—Heisel, M.: Problem-Oriented Documentation of Design Patterns. Fundamental Approaches to Software Engineering (FASE 2009). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 5503, 2009, pp. 294–308, doi: 10.1007/978-3-642-00593-0_20.

[10] Gao, T.—Ma, H.—Yen, I. L.—Khan, L.—Bastani, F.: A Repository for Component-Based Embedded Software Development. International Journal of Software Engineering and Knowledge Engineering, Vol. 16, 2006, No. 4, pp. 523–552.

[11] Kherraf, S.—Lefebvre, E.—Suryn, W.: Transformation from CIM to PIM Using Patterns and Archetypes. 19th Australian Conference on Software Engineering (ASWEC 2008), March 2008, pp. 338–346, doi: 10.1109/aswec.2008.4483222.

[12] Kirasić, D.—Basch, D.: Ontology-Based Design Pattern Recognition. Knowledge-Based Intelligent Information and Engineering Systems (KES 2008). Springer, Berlin, Heidelberg, Lecture Notes in Computer Science, Vol. 5177, 2008, pp. 384–393.

[13] Object Management Group: MDA, MOF, UML Specifications. 2013.

[14] Mapelsden, D.—Hosking, J.—Grundy, J.: Design Pattern Modelling and Instantiation Using DPML. Proceedings of the Fortieth Internet Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications (CRPIT '02), 2002, pp. 3–11.

[15] Meffert, K.: Supporting Design Patterns with Annotations. 13th Annual IEEE Internet Symposium and Workshop on Engineering of Computer Based Systems (ECBS 2006), 2006, doi: 10.1109/ecbs.2006.67.

[16] Navrat, P.—Bielikova, M.—Smolarova, M.: A Technique for Modelling Design Patterns. Joint Conference on Knowledge-Based Software Engineering (JCKBSE '98), IOS Press, 1998, pp. 89–97.

[17] Ó Cinnéide, M.—Nixon, P.: Automated Software Evolution Towards Design Patterns. Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE '01), New York, NY, USA, 2001, pp. 162–165.

[18] Rasool, G.—Philippow, I.—Mäder, P.: Design Pattern Recovery Based on Annotations. Advances in Engineering Software, Vol. 41, 2010, No. 4, pp. 519–526, doi: 10.1016/j.advengsoft.2009.10.014.

[19] Sabo, M.—Poruban, J.: Preserving Design Patterns Using Source Code Annotations. Journal of Computer Science and Control Systems, 2009, pp. 53–56.

[20] Vranic, V.—Snirc, J.: Integrating Feature Modeling Into UML. NODe/GSEM, 2006, pp. 3–15.

[21] Zamazal, O.—Svatek, V.: PatOMat – Versatile Framework for Pattern-Based Ontology Transformation. Computing and Informatics, Vol. 34, 2015, No. 2, pp. 305–336.

Peter Lacko received his Master's degree in software engineering in 2004, and his Ph.D. in artificial intelligence in 2009, both from the Slovak University of Technology. Currently he is Assistant Professor of informatics at the Slovak University of Technology. His research interests involve artificial intelligence, neural networks, software engineering and parallel and distributed computing. He is a member of ACM and IEEE and its Computer Society.

Peter Kajsa received his Master's degree in software engineering in 2009, and his Ph.D. in software engineering in 2013, both from the Slovak University of Technology. His main research interests include design and architecture of software systems, design and architectural patterns, model driven development, model driven architecture and other object management group specifications.

Pavol Nárat received his Eng. (Master's) degree cum laude in 1975, and his Ph.D. degree in computing machinery in 1984, both from the Slovak University of Technology. He is currently Professor of informatics at the Slovak University of Technology and serves as the Director of the Institute of Informatics and Software Engineering. During his career, he was also with other universities overseas. His research interests include related areas of software engineering, artificial intelligence, and information systems. He is a Fellow of the IET and a Senior Member of the IEEE and its Computer Society. He is also a senior member of the ACM. He serves in the Technical Committee 12 Artificial Intelligence of IFIP as the representative of Slovakia.