DOI:10.2298/CSIS091212032K

# Design Pattern Instantiation Directed by Concretization and Specialization

Peter Kajsa[1], Lubomir Majtas[1], and Pavol Navrat[1]

[1] Faculty of Informatics and Information Technologies, Slovak University of Technology, Ilkovičova 3, Bratislava, 842 16, Slovakia
{kajsa, majtas, navrat}@fiit.stuba.sk

**Abstract.** Design patterns provide an especially effective way to improve the quality of a software system design as they provide abstracted, generalized and verified solutions of non-trivial design problems that occur repeatedly. The paper presents a method of design pattern instantiation support based on the key principles of both MDD and MDA. The method allows specification of the pattern instance occurrence via the semantic extension of UML directly on the context. The rest of the pattern instantiation is automated by model transformations of the specified pattern instances to lower levels of abstraction. Such approach enables the use of higher levels of abstraction in the modeling of patterns. Moreover, the model transformations are driven by models of patterns besides the instance specification, and thus the approach provides very useful ways how to determine and control the results of transformations. The method is not limited to design pattern support only, it also provides a framework for the addition of support for custom model structures which are often created in models mechanically.

**Keywords:** Design patterns, concretization, specialization, MDD.

## 1. Introduction

There are many efforts to improve the quality of software system development or maintenance based on identification, acquisition and application of some kind of architectural knowledge [20]. In general, patterns are based on abstractions and generalizations of effective, reliable and robust solutions to recurring problems. Patterns provide abstracted, generalized and verified solutions of non-trivial problems. The concept of patterns was first introduced in the work of Alexander [17] dealing with urban solutions, but soon patterns were also defined and used in software engineering. The idea of applying verified pattern solutions to common recurring problems in the software design attracted considerable attention very quickly (cf. [4] and consequently, e.g. [5]), 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. Currently, design patterns represent an important tool for developers in the process of software design construction, and provide particularly effective ways to improve the quality of software systems. It is evident that design patterns are not the solution to all problems related to software development. Some have noted their limitations and propose new approaches to the knowledge representation in the software development domain, even proposing language architectures [21]. However, it is well known that the application of design patterns in software projects assists in the creation of modifiable, recursive and extensible software design [4]. CASE or other modeling tools provide nowadays 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 of instance integration into the context – application model. A more systematic approach to pattern instantiation in interaction between software designer and a supporting tool has been presented in [22].

Since patterns provide abstracted and generalized solutions to recurring problems, their application to a specific problem requires to concretize and to specialize the solution described by the pattern [5] (see Fig. 1).

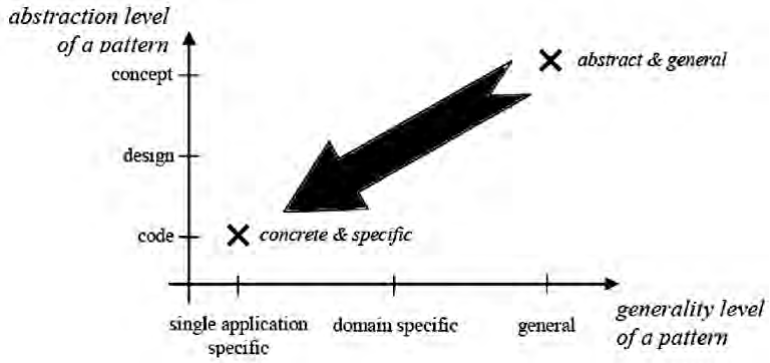


**Fig. 1.** Concretization and specialization of the solution described by the pattern, when the pattern is applied to a concrete and specific problem [5]

Specialization process of a design pattern typically lies in its integration into the specific context of the problem. The 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 itself. This is why this process is difficult to automate. Despite this, it is possible to make specializing of a pattern much easier by providing an appropriate mechanism for supporting application of design patterns.

The goal of 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, 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 of its structure. Majority of activities in the concretization process depends on a stable and fixed definition of the design pattern structure so that these activities are fairly routine. This is a good starting point for the automation of this process.

Consequently, we see a fairly good basis for the development of a method that would describe the way how to apply a design pattern based on supporting explicitly its specialization and concretization. We aim at proposing a method that would involve a specially devised tool supporting these two principal lines of design pattern instantiation.

Section 2 introduces several known approaches to tool based design pattern support and section 3 infers the open problems in this area. In the rest of the paper, we focus on the elaborated method of design pattern instantiation. Section 4 presents the theory about the method and it provides the method description. In the following section 5 the article covers particular aspects of method realization. Section 6 contains case study and the method evaluation. The paper is completed by a proposal of future works.

## 2. State of the Art

There exist several approaches which introduce their own tool-based support for pattern instantiation.

Mapelsden et al. [15] introduce an approach to design pattern application based on the Design Pattern Modelling Language. The authors describe this 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 a 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, the 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 into the pattern model. El Boussaidi et al. [11] present model transformations based on the Eclipse EMF and JRule frameworks. Wang et al. [12] provide similar functionality by XSLT-based transformations of models stored in XMI-Light format. Both approaches can be considered as driven by a single template and they focus mostly on the transformation process and do not set space for pattern customization.

Another method was introduced by Ó Cinnéide et al. [13]. 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). The 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. [8] 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. [9, 10] 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 [9], the implementation employs QVT based model transformations, and in the latter [10] 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. [14] 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. [16] 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 patterns in software designs and introduces a notation for the names of stereotypes: 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 the information.

## 3. Open Problems

The approaches that focus on the creation of pattern instances are typically based on the strict forward participant generation - participants in all roles are created according to a single template. Similarly, the support of design patterns available in traditional CASE or other modeling tools is usually based on UML templates of each design pattern. They are simply copied into the model with a minimal possibility for modification and integration in the rest of the model when pattern instance is created [1], [2]. However, patterns describe not only the main solution, but also many alternative solutions and variations. However, a developer is not allowed to choose an appropriate variant or a 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. Further, by the generation of source

code from a model with applied pattern instances, only class structure is generated, and the bodies of the methods of the patterns participants are empty. Consequently, the support of concretization has great deficiencies.

Moreover, the instance of a pattern created by a tool is typically without any connection to the rest of the application model. So the instance of a pattern has not been integrated into the application model, i.e. the context. It lacks associations 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 [15], the developer needs to model all pattern participants manually, and then to link these parts to the pattern model.

Our intention is to automate these activities. Our vision is that the developer simply specifies a pattern instance occurrence directly in the context, and the rest of the pattern structure is then automatically generated into the application model in an appropriate form.

## 4. Method Description

Our idea emphasizes collaboration between the developer and the CASE tool. We assume that we do not need to force the developer to explicitly model or mark all the pattern participants. Our aim is to encourage him/her just to suggest the pattern instance occurrence while the rest of the instantiation process is automated.

Patterns are often described as a collection of cooperating roles. Our approach is based on the idea [19] 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" while 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. Examples of domain dependent roles are presented in the Table 1.

The employment of patterns into the project allows the developer to think at a higher level of abstraction. When he decides to employ a pattern, the first thing he needs to take care of is how it will be connected to his project, how the solution will be integrated to the rest of his model / code. At this moment the developer does not focus on the entire pattern's inner structure, because it is irrelevant to him at this moment. The way how he integrates the pattern to the project lies in the specification of the domain roles. Their participants can be existing parts of the project or new ones 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 to the sample instance from the pattern catalogue.

**Table 1**. Examples of domain dependent roles of patterns [19]

| Pattern | Domain dependent roles | Description |
|---|---|---|
| Composite | Leaf and its Operations | Leaves and their operations provide all domain dependent functionality. Everything else is just the 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. |

When we look closer at such instantiation process from the perspective of its division into two more or less independent processes of specialization and concretization (described in the section 1 Introduction) [5], we can see that the user does the specialization process when he is specifying the domain roles. When he is supplementing pattern instance with the infrastructure roles he just completes the concretization process.

In our approach we do not want to replace the developer in the specialization process, but we want to relieve him of the necessity to instantiate the infrastructure roles during the concretization process. We want the developer to make a suggestion by the application of semantics as to where and which design pattern he wishes to be applied in the model and to specify the domain dependent roles. Then he can also specify which variant of the pattern to employ, and in what way he wants it to be generated. Subsequently, the rest of the pattern instance structure will be automatically generated by model transformations to lower levels of abstraction according to the instance specification.

In order to achieve the specified goal, it is necessary to provide an appropriate mechanism of pattern semantics in the application model. It is important to support insertion of semantics directly into the elements of the model, because such approach supports the specialization of pattern instances, and makes the creation of the instance specification effortless. Thanks to the semantics, the model transformations are able to understand the model of the application and recognize its parts.

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 to compare 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 the desired form obtained from the pattern model.

We note that model transformations automate the concretization process. They are driven by pattern instance suggestion and specification and by the pattern model as well. Such transformations have several capabilities. Firstly,

they provide a possibility to choose an appropriate configuration of the pattern by instance specification. Secondly, they enable the modeling of a custom pattern or structure by modification of the pattern model, and this way to achieve its generation into the model.

Moreover, our method assumes that the models and the transformations are split into more levels of abstraction in accord with the ideas of the MDA development process. These levels support work with instances of design patterns at various levels of abstraction. This process is shown in Fig. 2.

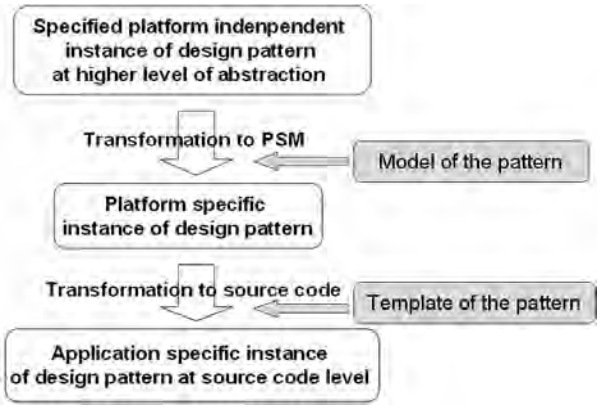


**Fig. 2.** Proposal of design pattern instantiation process

One of the main objectives of the approach is to consider ideas of model driven, iterative, and incremental development of software systems. It is important to note why the transformation to platform specific models (PSM) is necessary. It is at this level that the first differences in structure between instances of design patterns may occur. For example, some platforms allow multiple inheritance, others provide interfaces, etc.

## 5. Method Realization

The following subsections explain particular aspects of the method realization.

### 5.1. Realization of Pattern Instance Suggestion and Specification

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. We choose the semantical 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 can be applied directly to specific model elements such as Classes, Attributes, and Operations [6]. This way it is possible to specify participants of design patterns and relations between them directly in the context of the elements of the application model (for more details about the UML profile please see the section 5.3).

For example, Fig. 3 shows a suggestion of the Observer pattern instance via applying one stereotype <<Observes>> to a desired element, in this case, an association. From the 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 the information and the available pattern model and semantics, the transformation can recognize the other pattern participants need to be added to the model.

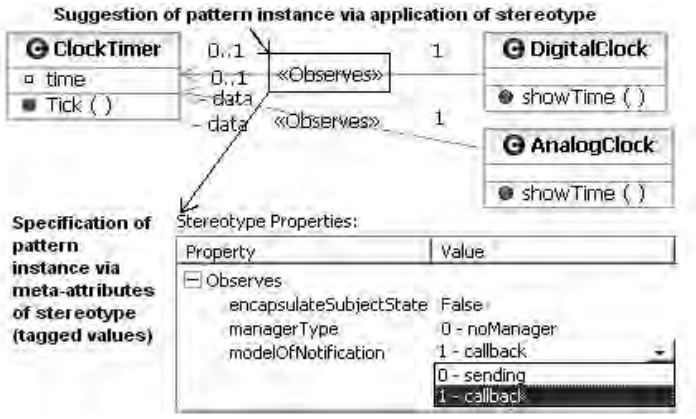


**Fig. 3.** Example of an application of the Observer pattern to a model. It represents a specified platform independent instance and thus the most abstract form of the Observer pattern instance

The transformation also needs information about how to generate the rest of pattern instance, e.g. variant of pattern, desired adjustments of 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 (Fig. 3). 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 the desired pattern can consist only of applying one suggestion mark – the stereotype onto the specified model element, when the developer wants the default variant of the pattern. Any other activities will be completed by a tool via model transformations. In this phase, developers do not have to concern themselves with the concrete details of the pattern structure, and

they can comfortably work with the pattern instances at a higher level of abstraction. The Application of the desired pattern is realized on elements of the system model or context, and thus the specialization process is supported.

### 5.2. Realization of Concretization Process

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 Fig. 3 is shown in Fig. 4. As it can be seen the transformation generates the rest of pattern structure in a desired form in accord with pattern suggestion and specification from Fig. 3. 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.

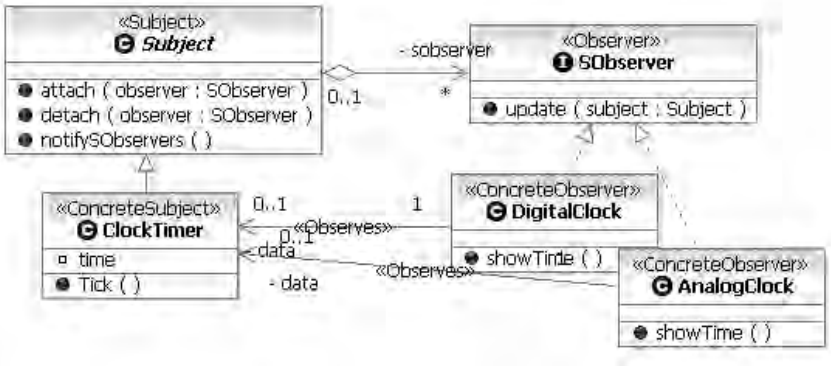


**Fig. 4.** The result of the transformation to Java target platform of the model from Fig. 3 in accord with the instance suggestion and specification

It is important that the transformation is realized and launched with a choice of target platform because, as mentioned earlier, at this point the first differences may occur in the structure of patterns depending on target platform. The choice of a target platform also determines the set of possible choices of data types before subsequent transformation to source code level.

As one can see in Fig. 4, 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 (more in Section 5.4). As a consequence, the instance is

clearly visible, and the developer can repeat the instantiation process at a lower level (PSM) directly from the optional second step, i.e. by 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.

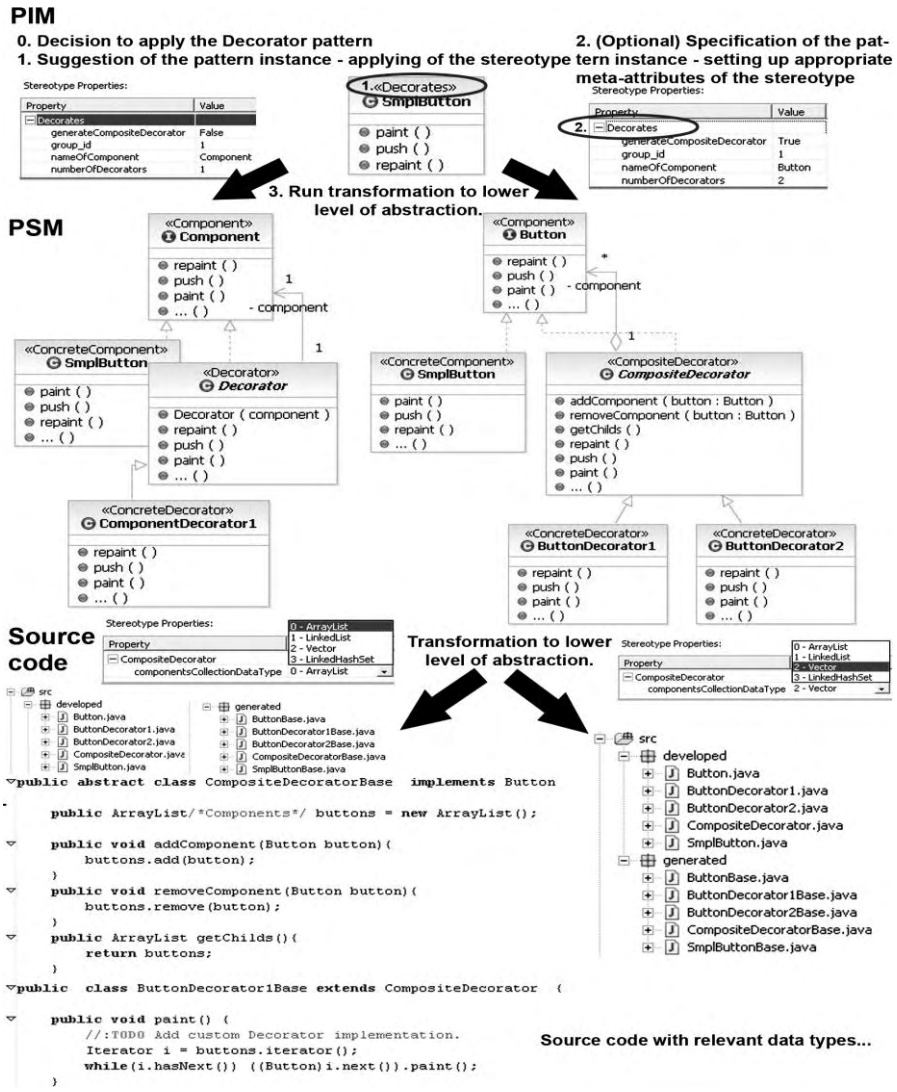


**Fig. 5.** An overall illustration of the pattern instantiation process

Two separate groups of classes are generated by the initial transformation to source code. The first is the base group which is always overwritten by subsequent source code generation. The second 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 it being overwritten.

Overall illustration of the described pattern instantiation process is shown in Fig. 5 using as an example a Decorator pattern application.

This way, our approach has achieved support for working with pattern instances at three different levels of abstraction:

- Pattern suggestion and specification level – PIM
- Design model level – PSM
- Source code level

### 5.3. Realization of UML Profile for Design Patterns

UML profiles provide a suitable way to define semantics for each design pattern and allow applying of semantics directly onto the elements of model. Consequently, a UML profile allows specification of participants of design patterns, and relations between them, directly on the elements of application model. The snippet of UML profile for Observer pattern is shown in Fig. 6.

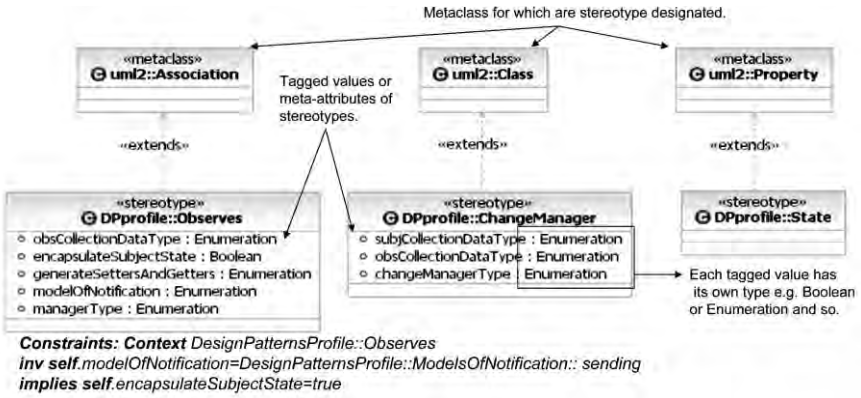


**Fig. 6.** The snippet of UML profile with some elements for Observer pattern

Authored UML profile provides semantics to various pattern instances adjustments, suggestions and specifications. However, it is not mandatory to apply all the semantics elements (stereotypes). The developer applies and specifies only what he needs to express. On the basis of applied semantics and pattern models with semantics, the transformation generates elements that are missing (more in the next Section 5.4). Because of the default values of meta-attributes of stereotypes, the transformation always has enough information for default behavior. Inconsistent specifications of pattern

instances are handled by OCL constraints which are part of UML profile as well (for example see Fig. 6).

Semantics of patterns is defined in one common UML profile for all supported patterns. However, the semantics of patterns from UML profile is not generalized for all patterns or structures. It contains semantics specific for patterns which are supported and in the consequence, when a developer wants to support new pattern or structure, he needs to add a semantics specific for this new pattern or structure into the profile (for more details see section 5.7 Extending of Support for New Patterns or Structures). It is important to remark, that it is not quite possible to create a profile generalized for all patterns or structures, because each pattern has its own semantics, purpose, variations and so on. Moreover, exactly our goal is to allow the developer to suggest and specify his intentions and design decisions in a specific way via semantics specific for the applied pattern. In case that the semantics applied by developer would be general for all patterns, intentions or decisions, we would not be capable of deducing some required specific information from such general semantics.

We tried to name the stereotypes according to the established names of pattern participants. However, a developer can change these names in the UML profile, but he must, of course, update also the pattern model.

Authored prototype of UML profile with description can be found in [26].

### 5.4. Realization of Transformations

Transformations performed by the tool 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 mark expresses the role of the element in the scope of the pattern. On the basis of this type of mark the tool is capable of creating mappings between models. The second type of mark expresses an association of the element with a variant of the pattern. On the basis of this type of mark 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 mark the following notation is defined:

```
[~]?StereotypeName::Meta-attributeName::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 model of the Observer pattern is exposed in the Fig. 7.

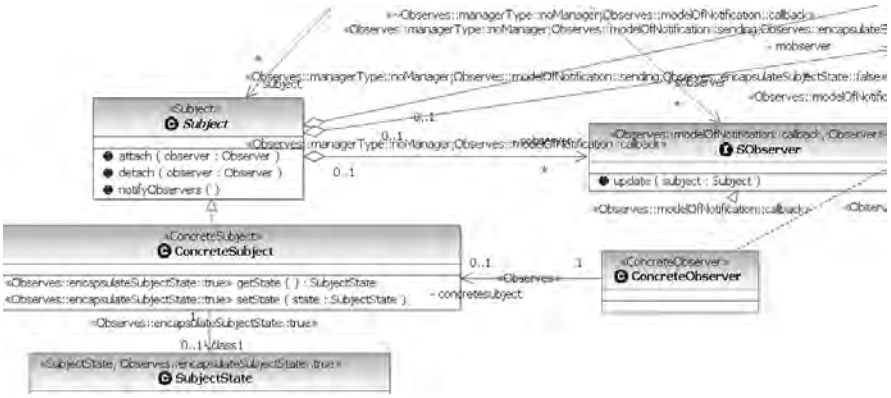


**Fig. 7.** Sample section of Observer pattern model by which the transformation is driven

The whole algorithm of the transformation is captured in the following Fig. 8.

The first action performed by the tool after the start of the transformation is the comparison of the first type marks in pattern model to the marks in the application model. When an instance of pattern is processed, only the marks with identical value of `group_id` are taken into consideration (for example, see `<<Decorates>>` stereotype in the Fig. 5 or case study in the Fig. 19 - 21). When the mark is without `group_id`, each next occurrence of the mark with the same name is considered as another instance participant. For example, stereotype `<<Observers>>` does not have `goup_id` meta-attribute and therefore, when the tool processes one of such marks the others are considered as other instances (for example, see example of Observer instantiation in the section 5.5).

Based on the first type marks comparison the tool is capable of making a 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 Fig. 3 in the previous section we have shown the application of the Observer pattern by applying one stereotype <<Observes>> on the directed association. From so marked association the tool can recognize that the parts Concrete Observer and Concrete Subject of this Observer pattern instance are present in the model already, and also which elements (in this case classes) in the application model represent these roles or parts.

Decisions about which variant of 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 the meta-attributes of stereotypes. These values are set up by the developer in the second step - specification of the pattern instance (see Sections 5.1 and 5.2 and Fig. 3 and 5).

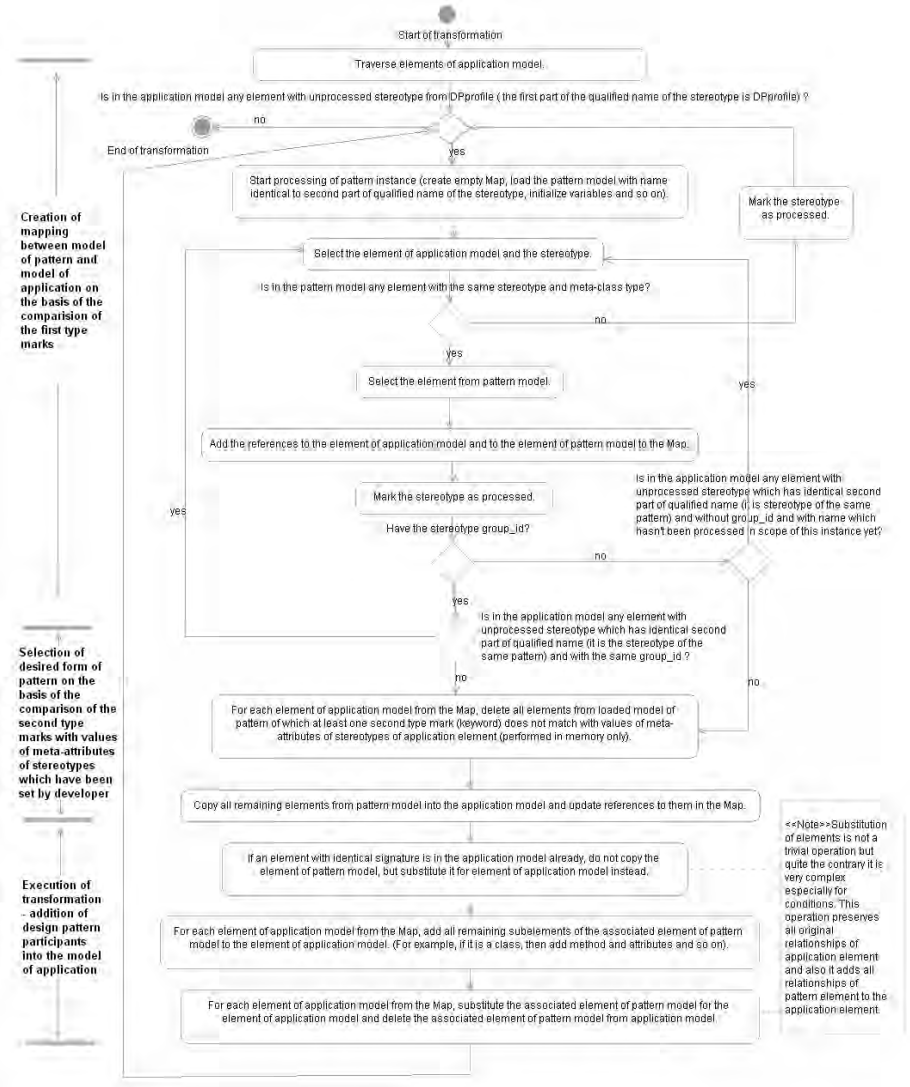


**Fig. 8.** Principles of tool functioning - tool under the hood.

After decision-making and selection of the desired pattern form, the alone transformation is performed. The results of the transformation are correctly specialized and concrete instances of the patterns created in the desired form, as presented in Fig. 4 and 5 in the previous sub-section.

Driving the model transformations by pattern models allows us to adjust results of transformations by modifying of the pattern models. 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

transformation, 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 code templates for now. Each pattern participant has own code template. The transformation takes code template with name identical to the stereotype name of the participant and it generates template's content into 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. The inconsistent states, such as duplicity of classes, illegal inheritance and others, are handled by the first transformation of the model of highest level of abstraction to the model of lower level of abstraction. The rules of correcting of such inconsistent states are common for all possible patterns or structures and therefore they are hard coded in the transformation algorithm. The transformation of the model to source code simply generates source code of each element from the model. An example of snippet of `Subject` code template is shown in the following Fig. 9.

The transformation to source code is still under our research. For more details see the section 7 Future Work. We have proposed the improvement of this transformation already.

```
 * (#)$0  'These file will be always overwritten'
 *
 */
public <% if(isAbstract){ %>abstract<% } %> class <%=className%> <%=extend%> <%=implement%> {

    /*generated fields*/

<% boolean getsubjectstate=true;
int pocet = 0;
for(int i=0;i<attributes.size();i++){
Property p=(Property)attributes.get(i);
if(p.getName().startsWith("ssobserver")){
    String typeOfState = subjectStates.size()>pocet ? (String)subjectStates.get(pocet) : "Object";
    pocet++;
    %>

    <%= JavaNameTool.getPropertySignature(p,collType) %>

    public void attach(<%=p.getType().getName()%> observer){
        <%=p.getName()%>.add(observer);
    }

    public void detach(<%=p.getType().getName()%> observer){
        <%=p.getName()%>.remove(observer);
    }

    public void notify<%=p.getType().getName()%>(){
        Iterator i = <%=p.getName()%>.iterator();
        while(i.hasNext()){
            <%=p.getType().getName()%> o=(<%=p.getType().getName()%>)i.next();
            o.update((<%=typeOfState%>)this.getSubjectstate());
        }
    }
```

**Fig. 9.** Snippet of code template of `Subject` participant of `Observer` pattern

### 5.5. Detailed View on the Method and the Tool in Action

This section provides illustration of the method and the tool, its functionality and usage by means of an example. The following Fig. 10 shows example of initial form of UML model before application of patterns.

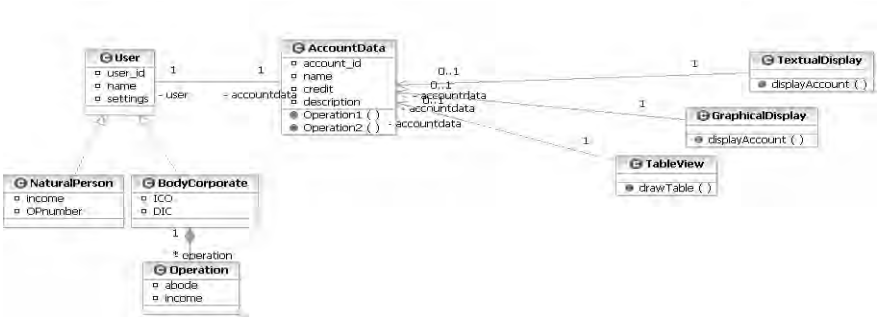


**Fig. 10.** Example of starting UML model before the application of patterns

The model represents an example of starting point of model into which the developer intends to apply, for example, Observer pattern now. In order to apply the desired pattern (in this case Observer) the developer suggests the instance occurrences via particular semantics marks – stereotypes (in this case stereotype `<<Observes>>`). Notice that the developer performs the suggestion of pattern instance occurrence on existing model elements directly in the context and so, in the consequence, the pattern instance will be integrated in the application model or context and thus there won't be necessary any manual specialization of pattern instance.

The resulting model after pattern instances suggestion is shown in the following Fig 11.

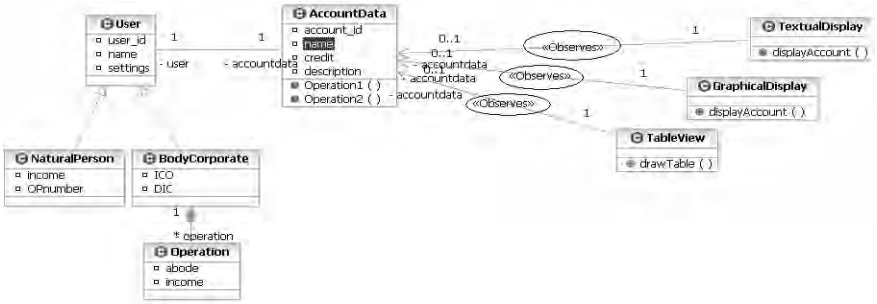


**Fig. 11.** The resulting model after pattern instances suggestion

It is important to remark, that each stereotype can be applied only on an instance of meta-class onto which is designated. For example, the stereotype

<<Observes>> extends the meta-class association and the stereotype <<Observer>> extends the meta-class class. Therefore, the tool does not allow to apply the stereotype <<Observer>> to any association or any other model element which is not an instance of meta-class class and also it does not allow to apply the stereotype <<Observes>> to any class or any other model element which is not an instance of meta-class association.

Now the tool knows what design pattern and where the developer wants to apply it. On the basis of comparison of this model to the pattern model by which the tool is driven, the tool also recognizes that the association between classes TextualDisplay and AccountData corresponds with association between ConcreteObserver and ConcreteSubject from the pattern model. The recognition is realized on the basis of first type of marks – stereotypes comparison in these models (see Fig. 12) and this way the tool creates mapping between these models.

Because the match of marks occurs on the association, the transformation recognizes that also the source and destination elements of associations (in our case ConcreteObserver and ConcreteSubject) must be already in the model of the application under development. In consequence, the transformation recognizes which elements of pattern model are in the model of application and which are not.

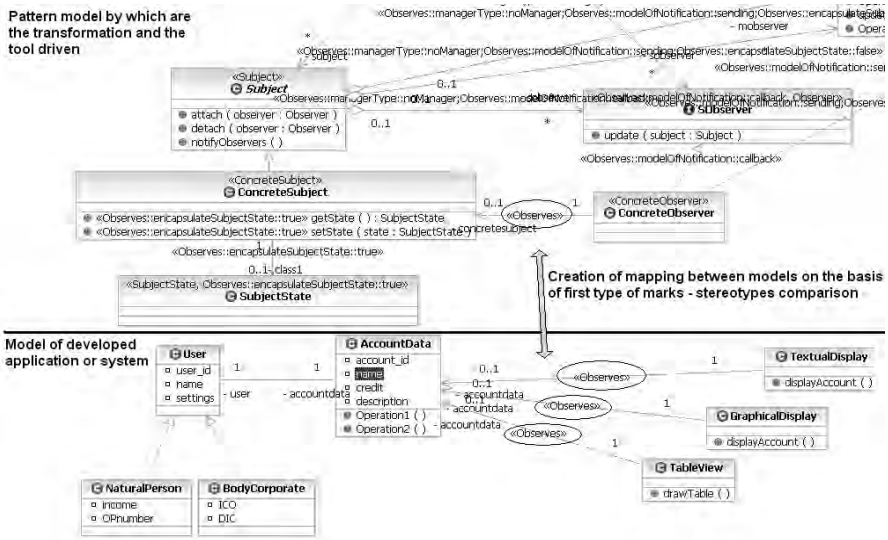


**Fig. 12.** Creation of mapping between model of developing application or system and pattern model by which the tool and the transformation are driven

Because the pattern model covers all the pattern variants, the tool needs to know which variant of pattern the developer wants to generate. In other words, the tool needs to know which of all identified missing pattern elements from the pattern model and what way it should generate into the model of

application. So the developer chooses the variant or modification of the pattern via setting up the values of particular stereotype meta-attributes in the next step of pattern instantiation (see Fig. 13). It is important to remark that the meta-attributes of stereotypes have set their default values. Therefore, this step is realized only if the developer wants to generate other than default variant of pattern. The possible variants and adjustments of pattern are defined in UML profile via enumerations or elements' primitive type specification such as boolean, integer and so on.

The developer specifies which variant or modification of pattern he desires and so the developer creates the specifications of suggested pattern instances. When the transformation is being executed, the tool processes all identified missing pattern participants from pattern model and it checks the second type of marks – keywords on these missing elements. As it has been introduced in previous section, for the second type of mark the following notation is defined (remind that these marks can be joined via ";", while the symbol "~" expresses negation):

```
[~]?StereotypeName::Meta-attributeName::value;
```

A missing element from the pattern model is generated into the model only when the specified meta-attribute of the specified stereotype has the specified value.

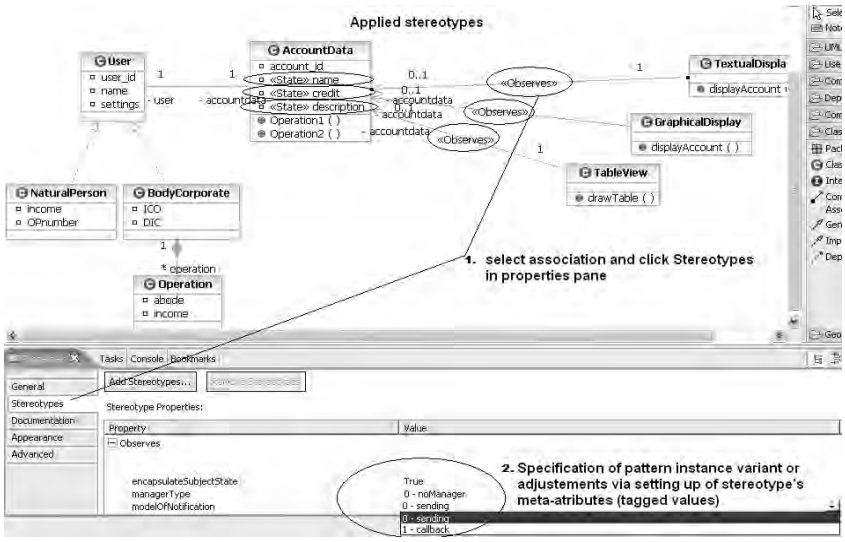


**Fig. 13.** Setting up of values of stereotype meta-attributes

Elements from pattern model of which at least one second type mark does not match 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 element

ConcreteSubject from the pattern model is identified as missing element in the application model, it 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 sets the value of meta-attribute encapsulateSubjectState of the stereotype Observes to true, because these methods are marked with the following second type mark <<Observes::encapsulateSubjectState::true>> (see Fig. 14, ConcreteSubject class of Observer pattern model).

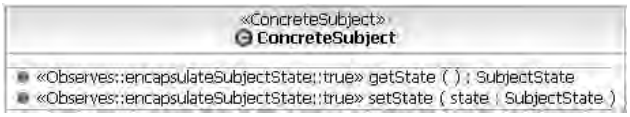


**Fig. 14.** Element ConcreteSubject from Observer pattern model

When suggestions and specifications of pattern instances are completed, the transformation can be launched simply from context menu of application model (for more details see user guide on [26]). The resulting model of transformation is shown in the following Fig. 15.

The following sample specification of pattern instances has been set in the second step of pattern instantiation by the developer (i.e. choosing pattern variant and adjustments via setting up the values of stereotype meta-attributes, see Fig. 13).

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

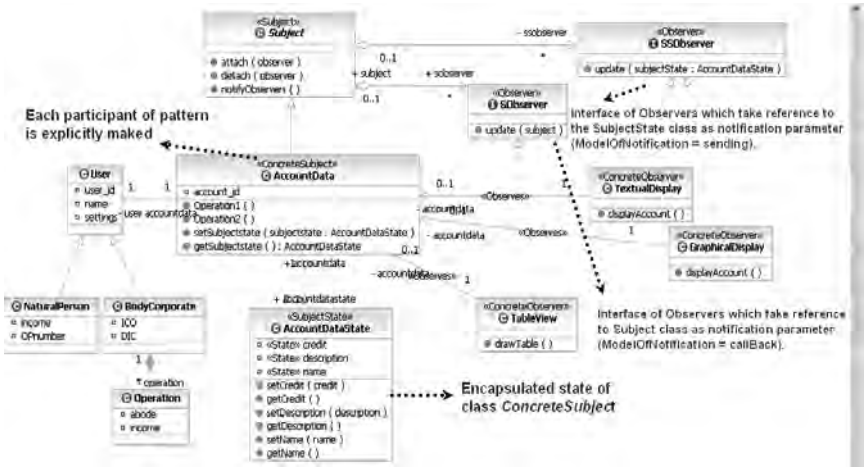


**Fig. 15.** The resulting model of transformation of model from Fig. 13

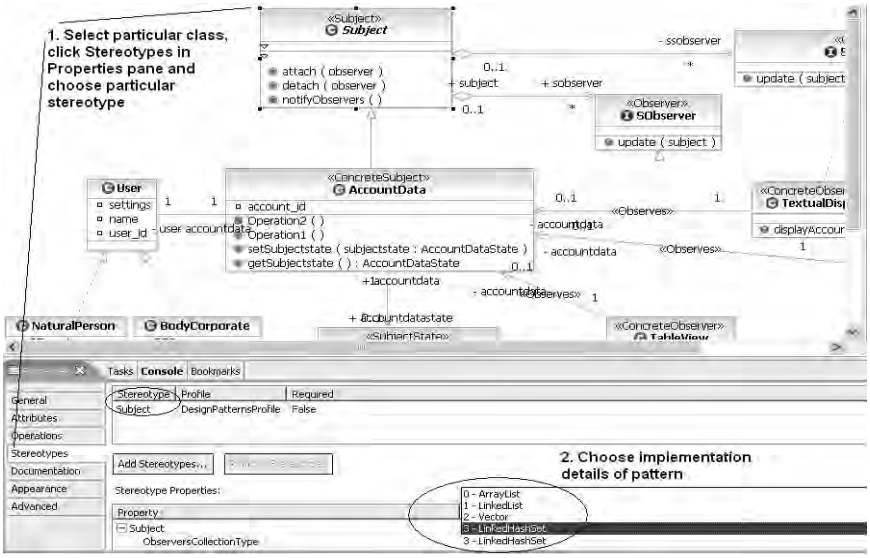


**Fig. 16.** Choosing of implementation details of pattern instances

The transformation marks explicitly also all the 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 second step and can specify implementation details of pattern instances directly without necessity of further stereotype application (see Fig. 16). 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 Fig. 16 to `Java` source code is shown in the Fig. 17.

The transformation to the source code generates two separate packages (`generated` and `developed`). The first is the base package which is always overwritten by subsequent source code generation. The second is the development package which is generated only by the initial transformation. The developer can write and add a specific implementation here without the threat of it being overwritten. Further, the distinct methods of observer notification have been generated for each group of `Observers` according to their specification (in our case `TextualDispaly` and `GraphicsDisplay` as the first group with `SSObserver` interface and `TableView` as the second group with `SObserver` interface, see Fig. 17). The transformation also uses chosen data types in the code generation. Description of source code generation has been introduced in the section 5.4. The snippet of code template of `Subject` participant of Observer pattern has been shown in the Fig. 9 as well.

After all, suggested and specified pattern instances from the highest level of abstraction have been transformed to the lowest level of abstraction – source code. The developer can utilize the created model and perform next iteration of development. For more details how the method and the tool work see user guide and video on [26].

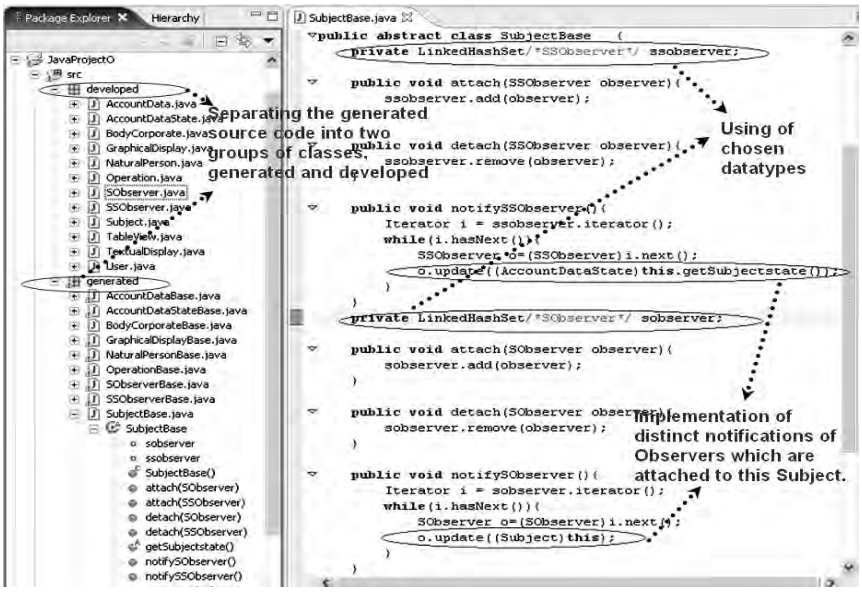


**Fig. 17.** The snippet of resulting source code of transformation of model from Fig. 16 to Java source code

### 5.6. Implementation

The presented method and the tool was implemented and verified in the form of an IBM Rational Software Modeler transformation plug-in. The following features have been implemented:

- Semantics in the UML profile for the patterns Factory Method, Decorator, Observer, Chain of Responsibility and Mediator
- Transformation of the highest level of abstraction (PIM) to the lower level (PSM) and transformation of PSM to source code
- Incremental consistency check mechanism
- Visualization of pattern instances and its participants
- Transformation of PIM to the lower level model PSM is driven by pattern models
- Models of design pattern covered all pattern variants and modifications which provide the basis upon which the transformational tool is driven
- Mechanism for adjustments of concrete form or desired variant of pattern instance for the patterns Factory Method, Decorator, Observer and Mediator

The first type of transformation of the highest level of abstraction (PIM) to the lower level (PSM) is 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 model of lower level of abstraction (PSM) to source code is implemented by frameworks JET, UML2 and EMF. The JET is also part of Eclipse Modeling Project in M2T (Model to Text) area. It provides infrastructure for source code generation based on code templates. The architecture of the implemented tool is shown on the following figure 18.

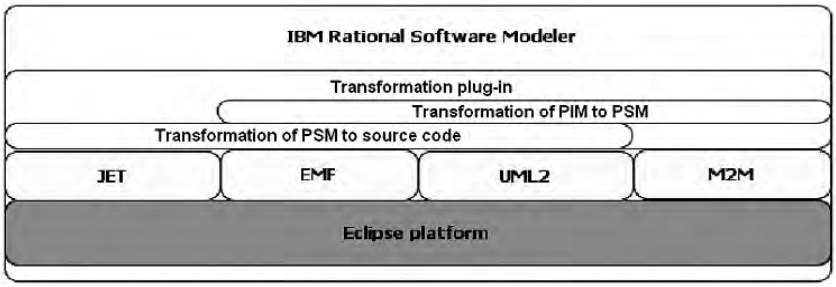


**Fig. 18.** The architecture of the implemented tool

### 5.7. Extending of Support for New Patterns or Structures

In order to extend the support for a new pattern or structure it is necessary to add definition of semantics of such new pattern into the existing UML profile. It is necessary to identify participants of a new pattern and to add definition of stereotype for each identified participant into the profile. All defined stereotypes should have the same second part of its qualified name (in RSM the stereotypes should have the same keyword). This part of the name represents the name of the new pattern. It is up to the developer how he names it, but the name should be unique in the set of names of supported patterns. After that it is necessary to identify variants of the new pattern and to create the according meta-attributes of the stereotypes (tagged values) and to create also definition of permissible values of the meta-attributes in form of enumerations or their type definition. If any stereotype can be applied in scope of one instance of a new pattern more than once, then the stereotype should have `group_id` meta-attribute in order to distinguish which stereotype belongs to which instance. In other words if cardinality of any participant of a new pattern is greater than one, then the stereotype of such participant should have defined `group_id` meta-attribute.

In the second step it is necessary to create a class model of the new pattern and to mark the participants with appropriate stereotype defined in the first step. Now the tool would be able to create mapping between models, because the developer places the same marks – stereotypes in the application model. So the tool can compare them simply. The tool still needs to know which participant it should generate and when. So it is necessary to add second type marks - keywords to the elements of class model of new pattern in introduced form:

```
[~]?StereotypeName::Meta-attributeName::value;
```

If the specified meta-attribute of the specified stereotype has the specified value, the element will be generated into the application model. Finally, it is necessary to export created model of a new pattern into XMI structure and place it into the working folder of the tool. The name of the file with the pattern model should be the same as the name of the new pattern (i.e. the second part of qualified name of stereotypes defined in the first step). The refresh or update of original UML profile is also necessary.

How the developer marks the model of the new pattern, thus the tool will generate the pattern into the model of application. So it is up to developer to mark the pattern model in the way that he desires. We do not want to restrict the developer. Our aim is to allow him to model any custom pattern. The tool simply takes a new pattern model, next the tool seeks in it the elements with marks identical to marks from application model placed by developer and then it maps the elements with identical marks. After that on the basis of the comparison of second type marks (keywords) from new pattern model and values of meta-attributes from application model which have been set by developer the tool filters out unwanted elements and it generates desired elements of the pattern. The tool performs all actions according to the

algorithm introduced in Section 5.4 (Fig. 8). That approach allows extension of transformation with new special functionality in form of definition of new rules and notations of marks. In this case the implementation of the new rules and the new notation recognition should be necessary, of course.

## 6. Evaluation

The presented method and its realization were evaluated in various experiments. In the following case study the aspects of correct pattern instantiation were considered in the evaluation process. The transformation algorithm (in Fig. 8) always checks on the presence of elements with identical definition by adding the pattern elements to the application model. Consequently, the transformation does not duplicate the pattern participants with identical definition when more instances of patterns are applied in the model. In addition, when the transformation of the model is run repeatedly, the incremental consistency of the model is verified. When an element with an identical definition is presented in the model, it is not duplicated. Instead, it is swapped. Illustrations of some case studies are shown in the following Fig. 19, 20 and 21.

The next evaluation was realized through experiments in which we have monitored and focused on the time of carrying out of an assigned task with and without usage of the tool. Also the count of generated and added source code lines has been observed. The tasks consisted of implementing specified instances of design patterns in a specified form. The average results of the experiments on a group of five programmers and five master degree students of software engineering are summarized in the Table 2.

**Table 2.** Average results of executed experiments

| Time with using the tool t1 | Time without using the tool t2 | Speed up t2/t1 | Number of generated code lines Ng | Number of added code lines Nd | Improving coefficient (Ng / Nd) + 1 |
|---|---|---|---|---|---|
| < 30 min | > 120 min | > 4 | 478 | 52 | 10,2 |

The quantity of the generated source code has been evaluated for each design pattern via metrics. The results of this evaluation are shown in Table 3.

**Table 3.** Quantity of generated source code

| Design Pattern | LOC | NOA | NOC | NOCON | NOIS | NOM | NOO |
|---|---|---|---|---|---|---|---|
| *Decorator pattern* | 223 | 9 | 6 | 7 | 11 | 103 | 22 |
| *Mediator pattern* | 212 | 6 | 6 | 7 | 9 | 50 | 15 |
| *Observer pattern* | 193 | 14 | 6 | 1 | 10 | 60 | 14 |

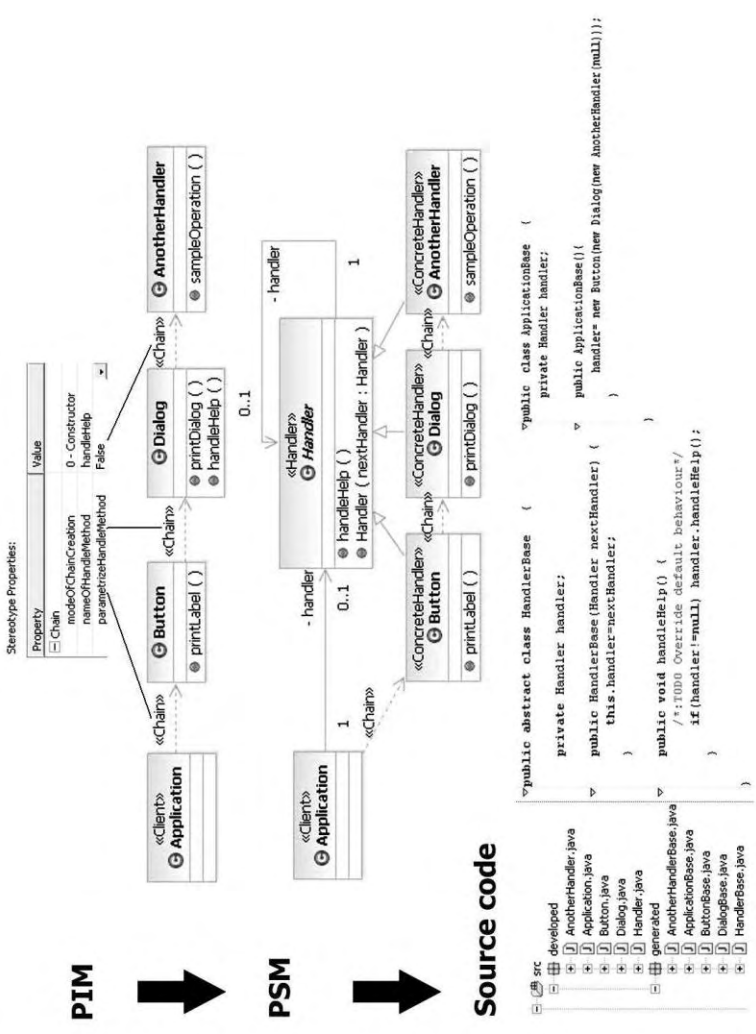


**Fig. 19.** Case study of simple Chain of Responsibility pattern instantiation

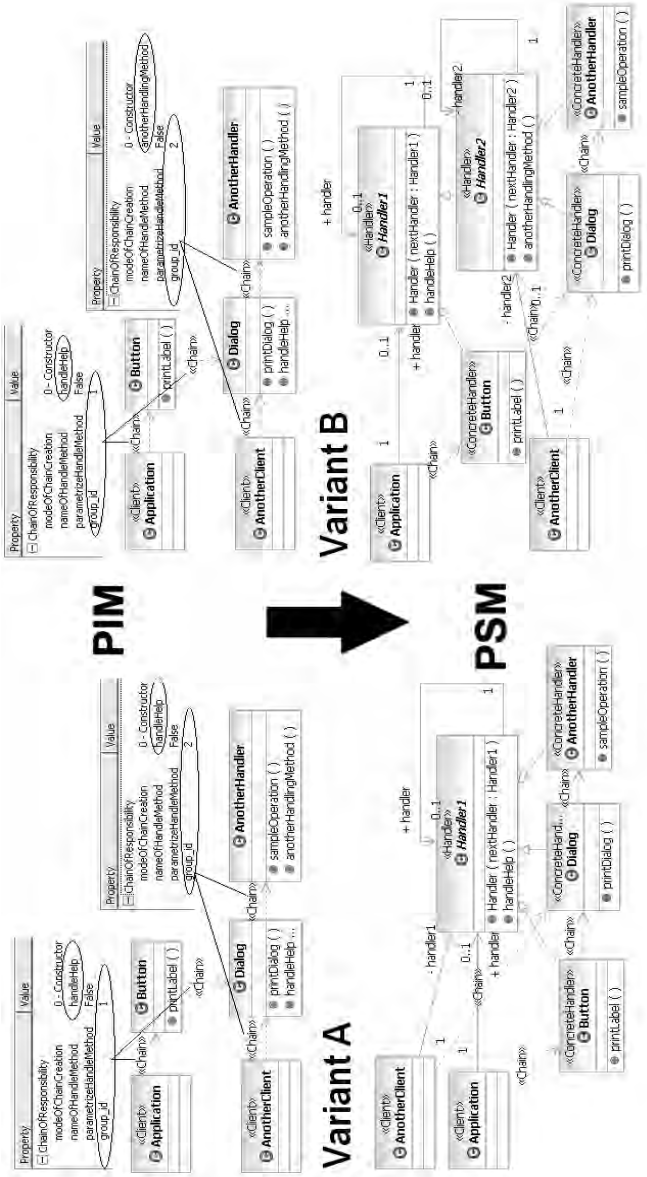


**Fig. 20.** Case study of advanced instantiation of Chain of Responsibility pattern. In this case, there are two clients. One client uses `Button` and `Dialog` as the processing objects and the other client uses `Dialog` and `AnotherHandler` as the processing objects. Variant B illustrates the case with different handled method names and Variant A with the same handled method names (in this case, all processing objects have super classes with the same definition, so the tool does not duplicate them, but it substitutes them instead)

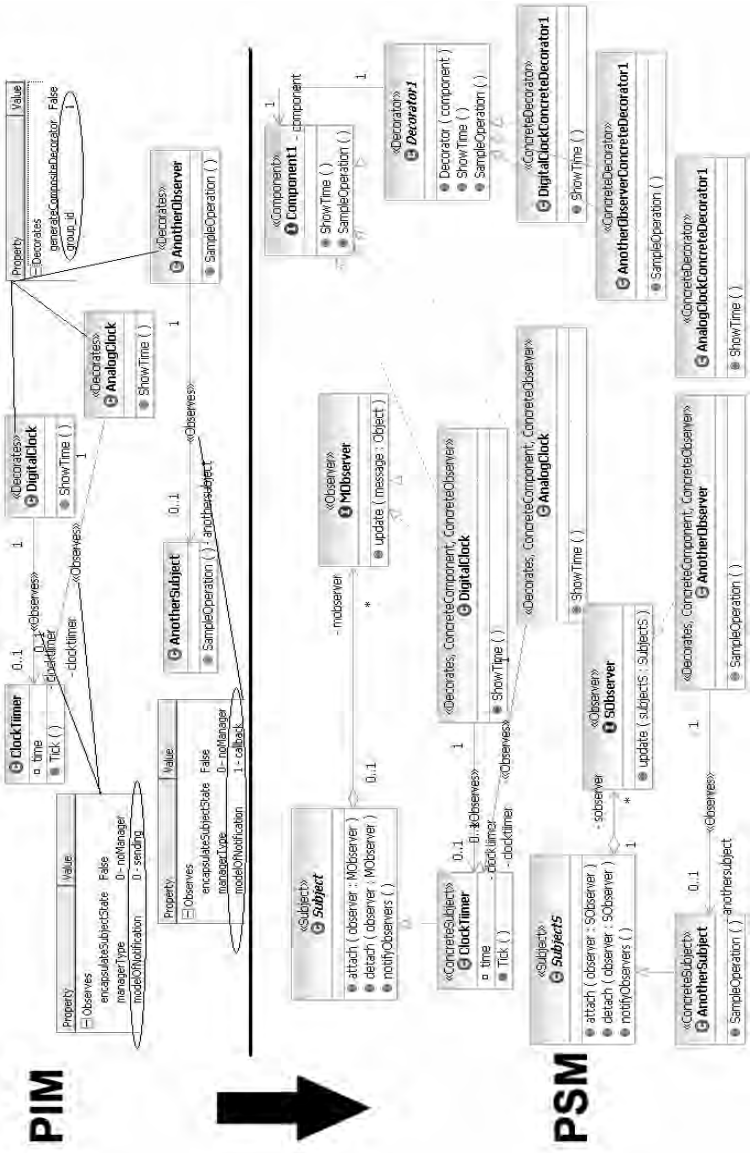


**Fig. 21.** Case study of sample Observer and Decorator pattern composition. Classes `DigitalClock`, `AnalogClock` and `AnotherObserver` have the same `group_id` and therefore they are considered as one Decorator instance. Moreover, the tool does not duplicate the elements with an identical definition, but it substitutes them successively as instance by instance are generated

Results of experiments show a significant improvement gained by use of the method and tool in the area.

## 7. Future Work

In the future, it is important to support also the fourth characteristic of the model driven development – the invertibility of models. The most important problem is to transform the source code to the design level (PSM), because the higher-level semantics cannot be reasoned directly and automatically from the source code in general. The knowledge is mainly available to developers and domain experts involved in the design process. Therefore, our aim is to add the missing semantics into the source code. Our idea is to mark explicitly and make visible higher-level (i.e. design) intentions in the source code via annotations. This way it would be possible to express also the semantics of patterns in the source code and the intention of annotated code as well. Consequently, it would be possible to expand the visibility of pattern instances from model into the source code by annotations. The pattern instances do not become invisible in huge amount of source code lines, quite the contrary, the full visibility of instances and their participants would be achieved by annotations. Consequently, using source code annotations the inverse transformation would be able to recognize pattern instance participants in source code and to transform them into a higher level of abstraction.

Besides this feature, also the traceability of transformations and pattern instances would be enhanced at the source code level. The code annotations make identifying of pattern participants in the source code quite easy. As a result, the tool based support of pattern instantiation or existing instances evolution, validation and identification at the source code level can be achieved in the form of code assists. Thanks to the annotations, the tool would be able to identify the pattern participants already implemented, and subsequently it would be able to offer to the developer the generation of any missing pattern participant or the possible evolution of instance in the given context. The evolution of existing instances of patterns without any tool-based support is quite difficult, because a developer has not a good vision about all concrete participants of pattern instances in the source code. However, this idea would bring significant improvement in pattern instantiation, evolution and validation in the source code.

Nowadays, we have proposed the improvement of the transformation to the source code. The method presented in this paper marks all pattern participants by stereotypes in the model. Our idea is that the transformation to the source code preserves the marking from the model and also extends it via annotations into the generated source code. Therefore no manual annotation of the code would be necessary in the generated source code, in comparison to the other present approaches [23, 24]. For more details about the improvement of the transformation to the source code and the method of

continuous support of the patterns at the source code level see our paper [25].

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

## 8. Conclusion

The abstraction, semantics and model transformations represent the key aspects of Model Driven Development and Model Driven Architecture. The possible level of the automation of the development process can be improved considerably thanks to them. The semantics applied in the models enables 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 the understanding of the model and its elements, it is possible to construct the transformation which transforms the model to a lower level of abstraction.

These principles represent the basis of the elaborated method of the design pattern application support. Thanks to the elaborated semantic extension of UML in form of UML profile, it is possible to specify participants of design patterns and relations between them directly on the elements of the application model. The suggestion and specification of pattern instances in the model allow the transition to higher levels of abstraction in the modeling of pattern instances. The instantiation details are split into more levels of abstraction, so developers do not need to concern themselves with concrete details of pattern structure at higher levels.

The transformations of models to lower levels of abstractions are driven by models of patterns. This aspect provides the key option to the developer to adjust the results of transformations by modification of these pattern models. This way it is possible to model any custom model structure and achieve support of its application to the model. Consequently, the method is not limited to GoF design pattern support only, but it also represents the framework of creation and addition of support for other custom model structures which are often created in models mechanically.

**Acknowledgments.** This work was partially supported by the Scientific Grant Agency of Republic of Slovakia, grant No. VEGA 1/0508/09 and by Slovak Research and Development Agency, grant No. APVV-0391-06 “Semantic Composition of Web and Grid Services”.

## References


1. Arlow, J., Neustadt, I.: Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML, Addison Wesley, (2003)
2. France, R., Dae-kyoo, K., Ghosh, S.: A UML-Based Pattern Specification Technique, pp. 193-206, IEEE transactions on Software Engineering, (2004)
3. Frankel, D.: Model Driven Architecture: Applying MDA to Enterprise Computing, Wiley Publishing, (2003)
4. Gamma, E. et al.: Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley professional computing series, (1995)
5. Návrat, P. et al.: A technique for modeling design patterns. Knowledge-Based Software Engineering - JCKBSE'98, pp. 89-97, IOS Press, (1998)
6. Object Management Group: MDA, MOF and UML Specifications. (2009) [Online]. Available: http://www.omg.org/
7. Borland Software Corporation: Borland Together Architect. (2009) [Online]. Available: http://www.borland.com/together/
8. Briand, L., Labiche, Y., Sauve, A.: Guiding the application of design patterns based on uml models. In ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance, pp 234-243, Washington, DC, USA, (2006). IEEE Computer Society.
9. Dong, J., Yang, S.: Qvt based model transformation for design pattern evolutions. In: J.-N. Hwang (Ed.): Proceedings of the Tenth IASTED International Conference on Internet and Multimedia Systems and Applications (IMSA 2006), Honolulu, Hawaii, USA, August 14-16, 2006. IASTED/ACTA Press 2006, 16-22.
10. Dong, J., Yang, S., Zhang, K.: A model transformation approach for design pattern evolutions. In ECBS '06: Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, pp 80-92, Washington, DC, USA, (2006). IEEE Computer Society.
11. Boussaidi, G., Mili, H.: A model-driven framework for representing and applying design patterns. In COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference, pp 97-100, Washington, DC, USA, (2007). IEEE Computer Society.
12. Wang, X.-B., Wu, Q.-Y., Wang, H.-M., Shi, D.-X.: Research and implementation of design pattern-oriented model transformation. In ICCGI '07: Proceedings of the International Multi-Conference on Computing in the Global Information Technology, Washington, DC, USA, (2007). IEEE Computer Society.
13. Cinnéide, M., Nixon, P.: Automated software evolution towards design patterns. In IW- PSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution, pp 162-165, New York, NY, USA, (2001). ACM.
14. Debnath, N.C. et al.: Defining Patterns Using UML Profiles. In IEEE International Conference on Computer Systems and Applications, pp.1147-1150, Washington, DC, USA, (2006). IEEE Computer Society.
15. Mapelsden, D., Hosking, J., and Grundy, J.: Design pattern modelling and instantiation using DPML. In Proceedings of the Fortieth international Conference on Tools Pacific: Objects For internet, Mobile and Embedded Applications, pp 3-11, Darlinghurst, Australia, (2002) ACM International Conference Proceeding Series, vol. 21. Australian Computer Society.
16. Dong, J., Yang, S.: Visualizing design patterns with a UML profile. In Proceedings of the 2003 IEEE Symposium on Human Centric Computing

Languages and Environments, pp 123-125, Washington, DC, (2003). IEEE Computer Society.
17. Alexander, C. et al.: A pattern language. Towns, buildings, construction. Oxford University Press, New York, USA, ISBN 0-19-501919-9, (1977).
18. Judson, S. R.: Pattern-based model transformation. In OOPSLA '03: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp 124-125, Anaheim, CA, USA, (2003). ACM
19. Majtás, Ľ: Tool Based Support of the Pattern Instance Creation. In: e-Informatica Software Engineering Journal, Vol. 3. Iss. 1, 89-102. (2009)
20. Havlice, Z. et al.: Knowledge Based Software Engineering. In: Computer Science and Technology Research Survey, elfa, Kosice, pp. 1-10, (2009)
21. Kollár, J., Porubän, J., Václavík, P., Forgác, M., Wassermann, L. et al.: New Generation of Language Architectures. In: Kollar, J. (Edt.): Computer Science and Technology Research Survey, elfa, Kosice, pp. 21-30, (2009)
22. Marko, V.: Template Based, Designer Driven Design Pattern Instantiation Support. In: LNCS 3255 – SOFSEM 2004, Springer-Verlag, pp. 144-158, (2004)
23. Sabo, M., Porubän, J.: Preserving Design Patterns using Source Code Annotations. In: Journal of Computer Science and Control Systems. (2009), pp. 53-56.
24. Meffert, K.: Supporting Design Patterns with Annotations. In: Proceedings of the 13th Annual IEEE international Symposium and Workshop on Engineering of Computer Based System. ECBS'06. IEEE Computer Society, Washington, DC, (2006), pp. 437-445
25. Kajsa, P., Návrat, P.: Design Pattern Support at Source Code Level Based on Annotations and Feature Models. In: Student Research Conference in Informatics and Information Technologies 2010, Bratislava, (2010), pp. 233-240
26. Kajsa, P., Majtás M., Návrat, P.: Web page of *Design Pattern Instantiation Directed by Concretization and Specialization*. (2010). [Online]. Available: http://www.fiit.stuba.sk/~kajsa/tool-based-design-patterns-support/

**Peter Kajsa** is a PhD student and a teaching assistant at the Faculty of Informatics and Information Technologies of the Slovak University of Technology in Bratislava. Peter received his Master degree in Software Engineering in the year 2009. 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. He has published several works in the area.

**Ľubomír Majtás** is a PhD candidate and a teaching assistant at the Faculty of Informatics and Information Technologies of the Slovak University of Technology in Bratislava. He received his Master degree in Software Engineering in the year 2006. His main research activities focus on design and architecture of software systems, Model Driven Development and Model-Driven Architecture where he specializes on support automation for design

pattern instances creation and their detection in the existing software. He has published several papers in the area.

**Pavol Návrat** received his Ing. (Master) cum laude in 1975, and his PhD. degree in computing machinery in 1984 both from Slovak University of Technology. He is currently a 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 abroad. His research interests include related areas from software engineering, artificial intelligence, and information systems. He published numerous research articles, several books and co-edited and co-authored several monographs. Prof. Návrat 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 and a member of the Association for Advancement of Artificial Intelligence, Slovak Society for Computer Science and Slovak Artificial Intelligence Society. He serves on the Technical Committee 12 Artificial Intelligence of IFIP as the representative of Slovakia.

*Received: December 12, 2009; Accepted: October 11, 2010.*