A Refactoring Tool for Design Patterns with Model Transformations

Zekai Demirezen[1], N. Yasemin Topaloglu[2]

Department of Computer Engineering, Ege University, Izmir, Turkey
[1]zekai@bornova.ege.edu.tr
[2]yasemin.topaloglu@ege.edu.tr

Abstract. Tools are essential for all phases of model driven development, especially for model transformations. We developed a tool to perform model refactorings with model transformations. Design pattern based model transformation is one of the applications of our tool. Our tool operates according to the source and target model definitions which are represented in UML. In the implementation of the tool, a transformation language was not used due to the comparison performed on the two models and the implicit usage of the transformation model by the transformation machine.

1 Introduction

Recently, model driven approaches in software development became one of the hot topics in software engineering community. The main motivation of model driven development is to increase the abstraction level in software development by considering the models as the primary artifact of software development and by focusing on model transformations as the main activity of software development. Model Driven Architecture (MDA) [1] is a framework that has been proposed by OMG to realize model driven development.

In a model transformation, elements of a source model are transformed to target model elements [1]. In order to realize the potential benefits of model driven development, tool support has considerable importance to provide automation on transformation activities [2].

France and Bieman [3] define two types of transformation as vertical transformation and horizontal transformation. In vertical transformations, a model is transformed to another model which is in a different abstraction level. In contrast, horizontal transformations do not change the abstraction level of the model and improve designs without changing the model behavior. Two approaches are defined to describe and implement model transformations [4]:

- *Mapping Transformation:* In this approach, each element from a source model is translated into zero, one or more elements in a target model.

- *Update Transformation:* In this approach, a model is modified by adding, deleting, or updating elements in the model. Model transformation that aims refactoring is a kind of this approach.

Another classification for model transformation approaches that is in parallel to the above classification is defined as *declarative* approach and *operational* approach [5]. Here, the declarative approach represents the mapping transformations and usually describes the transformations through rules. On the other hand, the operational approach represents the update transformations in which a sequence of actions to transform the input model to the output model is defined.

In this paper, we focus on update transformation approach and present our work to conduct horizontal model transformations with a tool that we developed for this purpose. Our initial aim was to provide automation in performing model refactorings, specifically design pattern based refactorings. However the design and the implementation of our tool enable to perform several model transformations.

The rest of the paper is organized as follows. In Section 2, we discuss the concepts that we use to perform model refactorings. In Section 3, we introduce the design of the tool and also discuss how the components of the tool cooperate. In section 4, we demonstrate the usage of our tool for the Observer pattern. Finally, we add some concluding remarks in section 5.

2 Background

Improving the internal structure of a software system while preserving its behavior is defined as refactoring [6]. Although most of the work on refactoring is about code refactoring, many researchers note that, a recent trend is to apply the concepts of refactoring at higher levels than code [7]. In our work, model refactoring is performed at the design level through model transformations.

Design patterns define proved experiences which can be used in constructing well-formed design [8]. A common practice in applying design patterns is to include patterns to the design models not in the initial design but in the later phases of design. Design patterns are considered as something that design evolves into and considered as a target for refactoring [9]. Therefore a transformation process on design models for model refactoring should take place.

In order to process models in transformation activities, meta-modeling is required. Meta-modeling defines the rules and the structures of models [10]. Role-based meta-modeling technique is proposed by France et al.[11] for design pattern meta-modeling. A role defines the tasks of the elements that form the system. With the definition of a role, structural and behavioral properties of the elements are provided.

In role-based meta-modeling, a design pattern is specified in three parts [11]:

- Structural Pattern Specification (SPS): It specifies the class diagram view of pattern solutions.
- Interaction Pattern Specification (IPS): It specifies interactions in pattern solutions.

- State Machine Pattern Specification (SMPS): It specifies a UML state chart diagram view of pattern solutions.

The approach used in design pattern meta-modeling is designed according to the four layer architecture of UML [10]. This architecture is developed by OMG to enable meta-model extensions. Four layers are defined as [10]:

- M3: The meta-metamodel layer contains the description of the structure and semantics of meta-metadata.
- M2: The meta-model layer contains the descriptions (i.e., meta-metadata) that define the structure and semantics of metadata.
- M1: The model layer contains the metadata that describes the data in the information layer.
- M0: The information layer contains the user data.

Design pattern meta modeling is suitable with this architecture and SPS, IPS and SMPS specifications are prepared in the M2 layer. In a model transformation, meta-models are produced for three concepts [12]:

- *Source pattern* defines the models that the transformation will be applied.
- *Transformation pattern* defines the transformation rules and the steps.
- *Target pattern* defines the output model of the transformation activity.

XMI is a standard language that supports metadata interchange between models [13]. XMI can be used to represent the source and the target patterns. The transformation pattern can also be shown with some extensions to XMI.

3 Design and Implementation of the Tool

The tool we developed aims to perform model refactorings for design patterns. Therefore we used the pattern specifications that are defined according to the role-based approach. In all definitions we consider structural and behavioral properties of meta-structures and their relations. All meta-definitions are compatible with the UML four-layer architecture.

According to our design, source pattern and target patterns specifications are used to derive the transformation meta-model by analyzing the source and the target patterns at run time. In the sub sections of Section 3, we discuss the architecture and the components of the tool.

3.1 Architecture

We employed the architectural design that is proposed by Wagner [14] in our tool. The architectural components of this design [14] are depicted in Figure-1. *Model editor* is a standard modeling tool such as Rational Rose that can be used to prepare

the application design models. *Rule editor* is used to prepare the source and the target meta-models. In fact, any CASE tool may serve as a rule editor and a model editor.

Transformation Preparer takes the application model and the source and the target meta-models as inputs and then matches the application model elements with the meta-elements. Also transformation meta-model is prepared by this component and given to the *Transformation machine* component as an input. *Transformation machine* executes the transformation steps defined in transformation meta-model. It performs the addition, deletion and replacement operations on model elements according to the transformation meta-model. And finally the result of this process is the transformed application model which is exported in XMI representation.

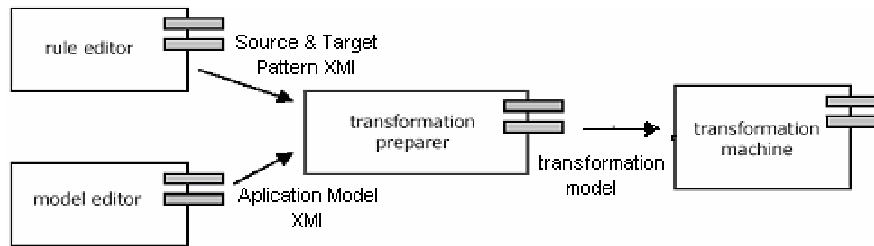

Fig.1. Transformation Process and Components of Architecture [14].

3.2 Transformation Preparer and the Transformation Machine

As described Section 3.1, all transformation activities are accomplished by the transformation preparer and the transformation machine in such an architecture. In our tool, the tasks of the transformation preparer and the transformation machine are handled by four sub-components. These sub-components are responsible of performing several sub-transformations. Below, these components are described in detail:

- Representation Transformer (RT): It transforms the meta-model's XMI representation to object representations by transforming XMI data to UML meta structures. As a result of this, XMI based definitions of the source and the target patterns are loaded to objects.
- Transformation Pattern Generator (TPG): This component produces transformation meta-model at run-time by analyzing structural differences between the source and target meta-models. Transformation meta-model is composed of *add, delete and update* operations. These operations are similar to Fowler's basic refactoring steps [6].
- Role-model Element Transformer (REM): This component matches the roles with application model elements. Matching activity is a kind of searching process of role's structural properties in the application model. Each role in M2 layer matches an element in M1 layer.
- Transformation Machine(TM): It is the last component that executes the transformation meta-model tasks on the application model. *Add, delete* and *update*

operations assign the role properties into the selected model elements. This activity enables to represent target model characteristics in the application model.

In Figure-2 the components of the transformation preparer and the transformation machine are shown. RT, TP and REM components compose the transformation preparer.

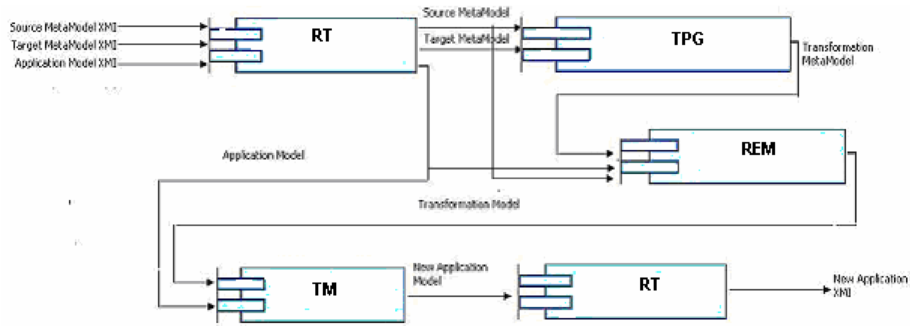

Fig.2. Components of the Tool.

3.3 Sub Transformations in the Components

There are four kinds of sub-transformations that are performed in the transformation preparer and the transformation machine. Components are designed according to transformation composition and reuse concepts. Transformations are reused through the specialization and the composition mechanism. The concept of a composite transformation is defined in Appukuttan [15].

To increase the capabilities of transformations, reusing existing transformations and composing further transformations from existing ones are needed. A composite transformation is formed of a parent transformation and a number of component transformations [15].

The sub-transformations are described below:

- XML Representation-Object Representation Transformation: The source, target and the application models in XMI are transformed into the objects representations. Figure-3 shows this transformation in the notation developed by Appukuttan [15]. In order to transform the package representation in XMI to the package object, first the model elements like c*lass, attribute, and association* are transformed into representations. Model element (ExM) transformation represents the re-use of the transformation steps in this transformation. While XMI Element-Package (ExP) transformation includes XMI Element-Class (ExC) transformation, it re-uses the ExM transformation to handle the sub step of the transformation.

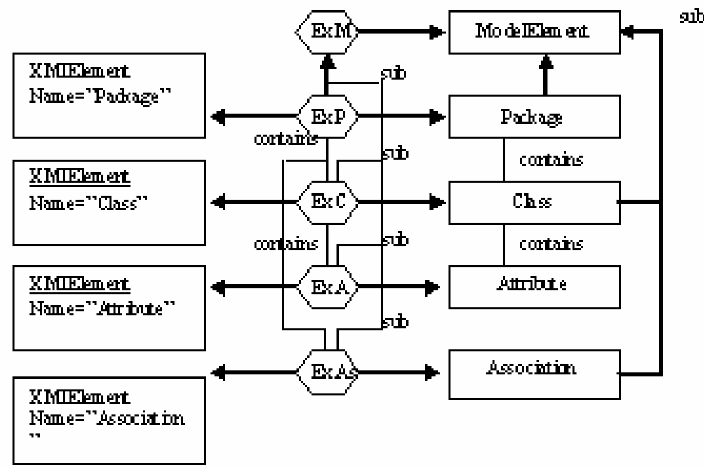

Fig.3. XML-Object Representation Transformation [15].

- MetaModel-Model Transformation: The roles constituting the target meta-model are matched with the application model elements and the source meta-model-source model transformation is the result of this matching. Source meta-model defined in M2 layer is transformed into the source model defined in M1 layer. In this transformation step, each role and sub roles which constitute these roles are assigned to the model elements in the application model. Figure-4 shows this sub transformation in the notation developed by Appukuttan [15].

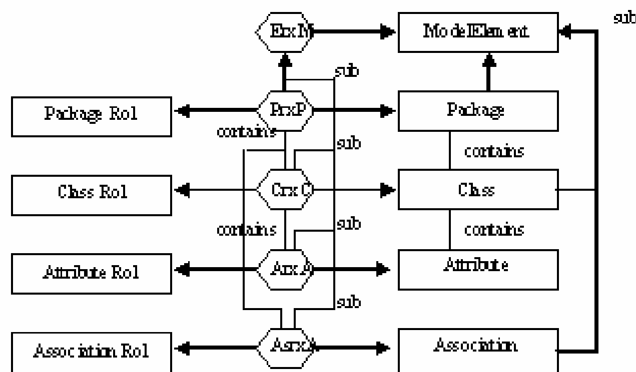

Fig.4. Role Model Element Transformation (MetaModel-Model Transformation).

- Application Model-Result Application Model Transformation: Model elements in the application model are transformed into the model elements of the target model by processing the steps in the transformation model. Transformation specifications in the transformation machine are applied on the application model in this transformation.
- Result Application Model Object Representation-XMI Representation Transformation: This step transforms the result model of the transformation process into the XMI format. Each model element is transformed into the corresponding element in the XMI format. Figure-5 shows this transformation proposed by Appukuttan [15].

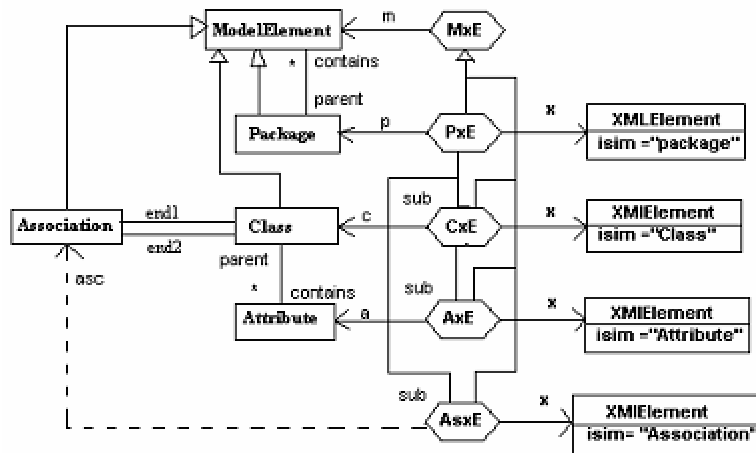

Fig.5. Object-XMI Representation Transformation adapted from Appukuttan [15].

We present how we applied the Appukuttan's composite transformation principle with the algorithm in Figure 6. This algorithm shows the transformation method that belongs to Classifier Class. In line 2, ModelElement's (super class of Classifier) transformation method is reused by calling super method. From line 4 to line 16, composite element transformations such as attribute, method transformations are implemented. In line 14 these composite element's transformation methods are just called. This method is an example of the adaptation of Appukuttan's tranformation reuse and composition principle in our tool.

```
1   public void applyPattern(ModelElement elem){
2           super.applyPattern(elem);
3                   Classifier nam=(Classifier)elem;
4           Iterator it=nam.features.values().iterator();
5           while(it.hasNext()){
6                   ModelElement elemTar=(ModelElement)it.next();
7                   if(elemTar.delete)
8                           deleteList.add(elemTar);
```

```
9                         else{
10                                  ModelElement
elemApp=((ModelElement)features.get(elemTar.getID()));
11                                  if(elemApp==null){
12                                           addList.add(elemTar);
13                                  }else{
14                                           elemApp.applyPattern(elemTar);
15                                  }
16                        }
17              }
18     }
```

Fig. 6. Composite transformations for Classifier Class.

4 Application of the Tool

4.1 Case Study: Observer Design Pattern

In this section, we demonstrate how our tool can be used to perform refactoring for the Observer [8] pattern.

The *Observer* pattern describes the situation in which a one-to-many dependency between objects is present. In this case, when one object changes state, all its dependents are notified and updated automatically.

To implement the transformation for the Observer pattern to an application model, following steps should be performed:

- Preparing the Observer Source Pattern Specification: The source pattern for the Observer pattern can be prepared with a Rational XDE tool. We consider only the structural parts of the design pattern. Each design pattern is defined with a standard format in design patterns catalogue [8]. Applicability and motivation sections of the patterns in the catalogue are problem definitions for patterns. We provide a definition of the *Observer* pattern according to XMI. In some parts we use stereotypes in order to provide problem semantics. During stereotype usage, we prefer to use universal standard stereotypes. For example *Boundary, Controller, Model* [16] stereotypes enable us to define semantic parts of the meta-models. In Figure-7, simple source pattern for Observer is shown in the UML notation

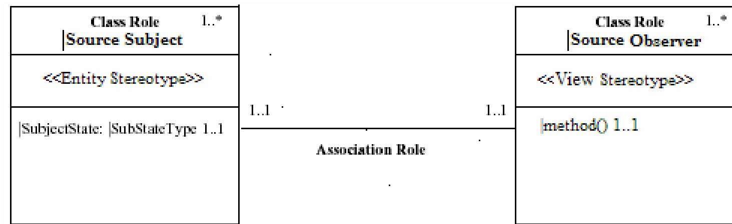

Fig.7. Observer Source Pattern SPS.

- Preparing Observer Target Pattern Specification: This activity is similar to the source pattern preparation. The target specification is the solution of the design pattern. We use the *Class structure* and the *Collaborations* sections of the Gamma [8] catalogue to derive this specification. Semantic parts of the design pattern target specification are supported by the stereotype usage. In Figure-8, simple target pattern for Observer is shown in the UML notation.

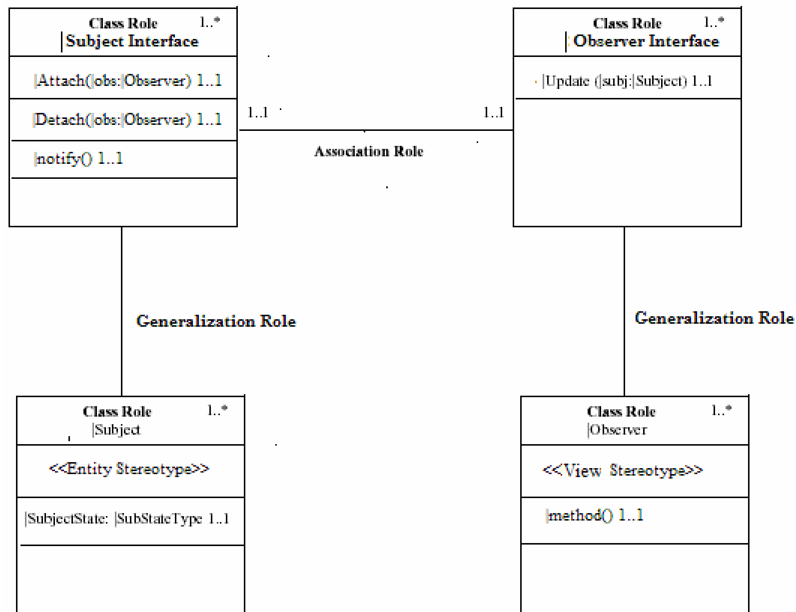

Fig.8. Observer Target Pattern SPS.

- Application of the Tool: After the rule definition activities, the tool is executed for transformation with the application model. Observer source and target definition is sufficient for observer design pattern based transformation. In Figure-9, a sample application model is shown. For simplicity, UML notation is used. This model is used for describing the inner activities in this part.

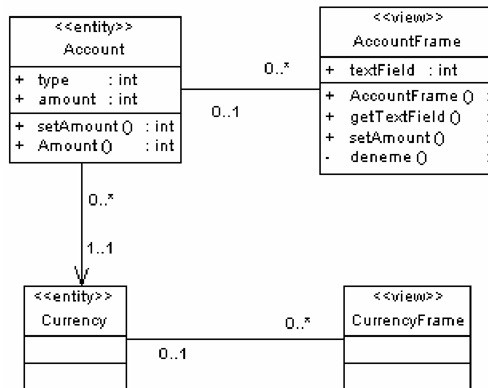

Fig.9. Sample Application Model

REM component of the tool transforms the source and the target patterns to the Source and the Target models. In Figure-10 and Figure-11, these models are shown for our sample model.

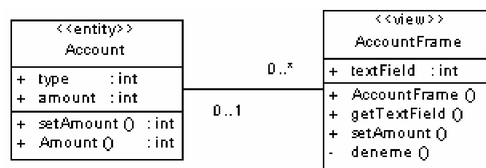

Fig.10. Source Model

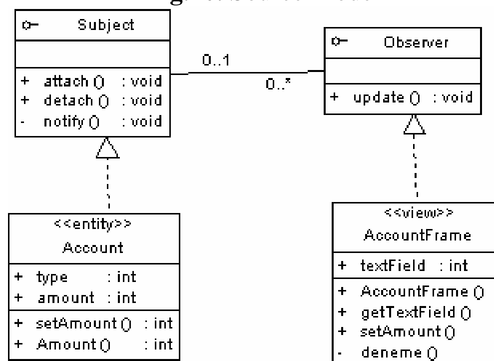

Fig.11. Target Model

TPG Component takes these two models and prepares transformation model with using add, delete, update and unchanged labels. In Figure-12, generated transformation model for our sample model is shown. Transformation model is executed by the TM component and the result application model is produced.

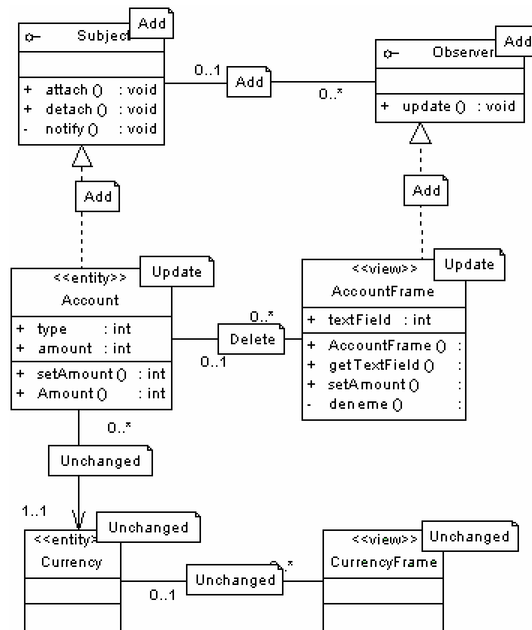

Fig.12. Transformation Model

User interacts with the tool in some parts during the execution. First of all, user selects targeting design pattern from pattern list and gives the application model XML file as an input. Screen1 is used for these selection operations as seen in Figure-13.

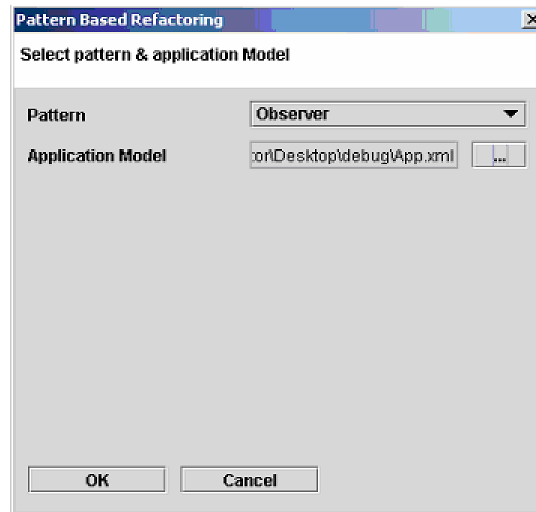

Fig.13. Screen1

After that, the window which shows the roles and their possible model element player list is presented. User selects one of the elements from the list. So the tool can

assign the role properties to the selected element. If the user does not want to assign any element from the application model or if there is no element that may play that role, tool can create a new element. This new element has the same properties as the role. User operates Screen2 for this purpose as shown in Figure-14. This screen iterates until all the roles are assigned to one of the application model element or to a new element. Finally user exports the new application model from the tool as XMI file.

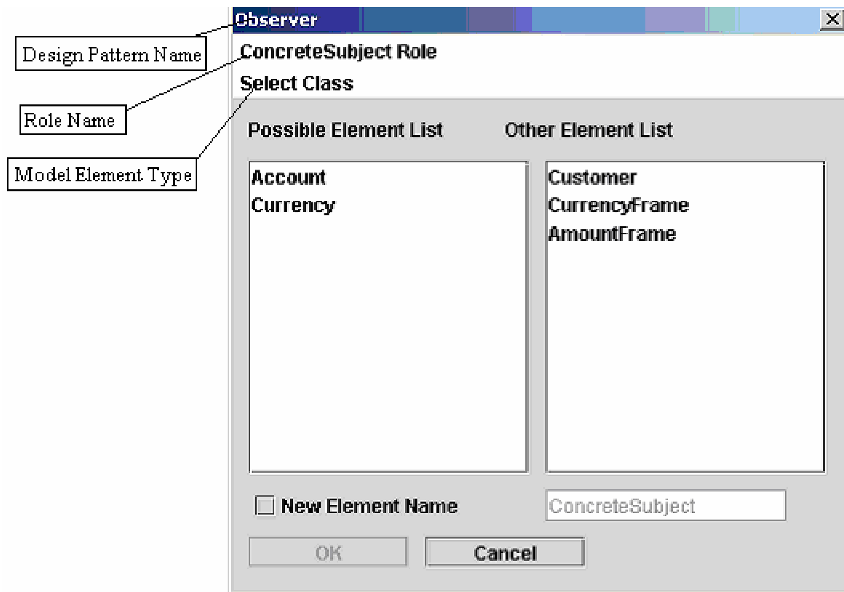

Fig.14. Screen2

4.2. Other Design Patterns

Our tool's execution principle can also be used to perform refactoring for other design patterns. For this purpose, design patterns IPS definitions must be available. Below we present source metamodel definitions for some design patterns briefly.

FACTORY METHOD:

In order to apply the *factory method* design pattern, with using IPS design models, a method that includes an instantiation of other classes can be searched.

COMPOSITE:

First, part-whole structures should be searched within the original design. If any class composite other classes and any client has an association with the single element and the composite element then refactoring for the *composite* pattern can be performed.

FACADE:

In order to perform refactoring for the *façade* pattern, in the original design, there must be a client which access (association with) many classes within the same package.

ADAPTER:

For the *Adapter* pattern, searching activity should be focused on finding a class that has a different interface. In an IPS model, interaction that includes the "instance of" style operator shows that the interaction behaves as a different interface. For these kind of interactions, refactoring for the adapter pattern can be done.

5 Conclusion

In this paper, we introduced a tool that performs horizontal transformations on design models. Our focus is on model refactoring through update transformations within the context of model driven development. In our work, the static role model definitions constitute the model query part. In the transformation process, the transformation operations are realized at run-time by the comparison of the source and target patterns instead of by using static definitions coded in a model transformation language.

Formal definitions of sub-transformation activities are not completed yet. Due to the limitations of XMI we are not able to define the full definition of the transformation patterns. Definition of semantic part of design models can only be given with stereotypes in our implementation.

The common use of design patterns and the need of automating the development of pattern based software development is the main motivation of our study. We investigate the possible contributions of automating these processes with model driven concepts.

This project has been implemented as a part of a Masters Thesis [17]. In our future research we aim to complete the definitions of other design pattern transformations. In addition to static pattern specifications, we are planning to add IPS support to the tool in our future work.

References

1. OMG, 2003a, MDA Guide Version 1.0.1, Available from www.omg.org.
2. Selic B., "The Pragmatics of Model-Driven Development," IEEE Software, Vol. 20, sayfa 19-25, 2003
3. France R. ve Bieman J., "Multi-view Software Evolution: A UML-based Framework for evolving Object-Oriented Software," Proceedings International Conference on Software Maintenance (ICSM 2001), 2001.
4. Porres I., "Model Refactorings as Rule-Based Update Transformations," Proceedings Unified Modelling Language Conference 2003.
5. Metzger, A. "A Systematic Look at Model Transformations", in Model Driven Software Development, (Eds.) S.Beydada, M.Book and V.Gruhn., Springer-Verlag, 2005.

6. Fowler M., Beck K., Brant J., Opdyke W., Roberts D., Refactoring: Improving the Design of Existing Code, Addison Wesley, First Edition, 1999.
7. Mens, T. And Tourwe, T., “A Survey of Software Refactoring”, IEEE Transactions on Software Engineering, Feb. 2004, pp 126 – 139.
8. Gamma E., Helm R., Johnson R, Vlissides J., Design Patterns: Elements of Reusable Object-Oriented Software; Addison-Wesley Professional Computing Series, 1994.
9. Astels, D., “Refactoring with UML”, Proc. 3rd Int’l Conf. eXtreme Programming and Flexible Processes in Software Engineering, pp. 67-70, 2002.
10. OMG, Meta Object Facility (MOF) Specification, The Object Management Group, 2000, www.omg.org
11. France R., Kim D., Ghosh S., ve Song E., “A UML-Based Pattern Specification Technique,” IEEE Transactions on Software Engineering, Vol.30, No.3, 193-206, 2004.
12. Judson S., France R., “Model Transformations at the Metamodel Level,” Proceedings Workshop in Software Model Engineering,” Unified Modeling Language ’03 Conference, 2003.
13. OMG, XML Metadata Interchange Specification 2.0. OMG Document 99-06-05. www.omg.org
14. Wagner A., “A Pragmatical Approach to Rule-Based Transformations within UML Using XMI.difference,” Proceedings of the Workshop on Integration and Transformation of UML models (WITUML 2002), 2002
15. Appukuttan K., Clark, T., Reddy S., Tratt L., Venkatesh R., “A Pattern based model driven approach to model transformations,” Presented at Metamodelling for MDA 2003, York, UK, November 2003
16. Jacobson I., Christerson M., Jonsson P., Overgaard G., Object Oriented Software Engineering, A Use Case Driven Approach, Addison-Wesley Publishing, 1994, ISBN 0-201-54435-0
17. Demirezen Z., “Application of Design Patterns and Tool Development”, Msc Thesis, Ege University, 2004