Using Patterns in Domain-Specific Languages

Tamás Mészáros, Gergely Mezei, Tihamér Levendovszky

Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Goldmann György tér 3, H-1111 Budapest, Hungary
mesztam@sch.bme.hu, {gmezei, tihamer}@aut.bme.hu

Abstract: Time spent on development and the quality of the resulted application has always had a natural key importance in software development. Raising efficiency can be achieved in several ways: one of them is to use modelling to be able to solve the problem at a high abstraction level. Modelling is especially effective when applying metamodelling and creating Domain Specific Languages, which help us to handle business rules in their own environments. Another possibility is to use patterns – customizable reusable components – both in the modelling and in the implementation phase. The aim of this paper is to present a possible solution for applying these two facilities at the same time: giving the possibility to create general but customizable model-patterns. Creating domain specific patterns and reusing them in other domain specific models offer great perspectives for rapid application development and keep reliability at a high level as well.

Keywords: modelling, metamodel, DSL, design patterns

1 Introduction

Nowadays, proper modelling is gaining a constantly growing importance in software development. A modell represents a simplified copy of a real system by neglecting unimportant details and letting the developer focus on real business processes. Models often use graphical visualization to describe entities and their relationships, thus, they enable designing systems at a high abstraction level by using graphical modelling environments.

Metamodelling means describing a modelling language by models. This means that metamodels are the modell of other models. A metamodell specifies the elements available in the instance models, the attributes assignable to the elements, as well as the relationships between these elements and the properties of these relationships. Metamodelling enables creating various high level modelling languages which fit the modelled area the best. These high-level modelling languages are called Domain Specific Languages (DSL), as they facilitate creating models in a specific domain. While, for example, the UML [1] language family is

a set of general purpose modelling languages. UML languages can be used in various situations, but their universality makes it difficult to apply them in some cases compared to DSLs.

The other main topic of this paper – beside domain specific modelling – is the pattern handling in modelling. A pattern is a reusable entity, which describes a frequent design or implementation problem, and gives a general, but customizable solution for it. Illustrative examples are design patterns defined by a UML class diagram [2]. Patterns born as best practices, which give the best solution in certain environments, later they become formalized, documented and will be made available for others. The aim of this paper is to extend and generalize the idea of patterns to Domain Specific Languages and give an effective possibility to create, organize and apply them in metamodelling environments.

This paper is organized as follows: First we introduce the applied modelling environment followed by the description of already existing solutions. In the contribution part, we introduce our solution, and in Section 5, we present a real world case study to demonstrate the applicability of our solution. Finally conclusions are drawn and future works are outlined.

2 Background

The research presented in the paper is based on an n-layer metamodelling framework, Visual Modeling and Transformation System (VMTS) [3]. In VMTS, there is no restriction for the number of metamodel-instantiation levels. Models are stored as directed and attributed graphs, where the nodes correspond to the entities of the modell and edges represent the relationships between them. VMTS provides the possibility to create and edit metamodels, and design models by instantiating metamodels. It also allows declaring constraints on models just as to apply various modell transformation techniques to them.

The architecture of VMTS is illustrated in Fig. 1.

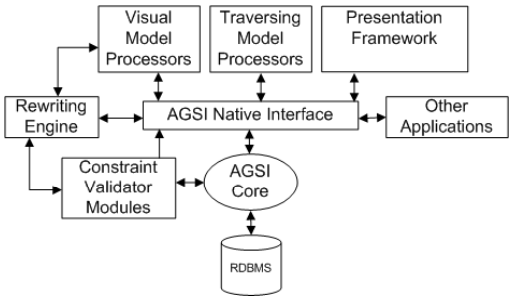

Figure 1
The structure of VMTS

We will use two main components of VMTS studying our solution: the Presentation Framework and the AGSI Native Interface. VMTS stores modell information in a relational database. AGSI Native Interface is a Data Access Layer for this database. AGSI is used to create and edit models. The Presentation Framework (VPF) is the graphical environment part of VMTS used for displaying and editing the models with their proprietary representation [4]. VPF facilitates assigning plug-ins to metamodels, these plug-ins are responsible for custom model-visualization and behaviour.

There already exists an external component for VMTS with the goal of supporting creation, modification and insertion of patterns. The name of this component is Pattern Factory. However it supports editing design patterns in case of UML class diagrams, it does not support patterns for DSLs, and it does not have a graphical design interface. We do not have any influence on final element layout either.

3 Related Work

There are numerous implementations and industrial applications, which make use of patterns.

The Rational XDE environment [5] enables designing programs through a graphical UML modelling framework, where we have the possibility to insert traditional UML design patterns with high customization facilities. The lack of this implementation is that it is restricted to UML class diagram elements only since the modelling framework supports this language exclusively.

VIATRA 2 [6] is a generic modelling and modell transformation framework which is integrated into the Eclipse platform. Patterns in VIATRA 2 can be applied to any domain specific language created with framework, but these patterns can be used only in connection with graph transformations not in general modelling purposes [7]. Furthermore, it does not provide a graphical interface to create and modify patterns.

Other commercial applications like Borland Together [8] or Rational Software Architect [9] have the capability to define and apply patterns in a friendly and productive environment, but these tools are also restricted to use UML only.

4 Contributions

The task of handling patterns in DSLs consists of four parts: (i) creating, (ii) storing, (iii) organising and (iv) applying patterns.

The earlier implementation of pattern handling in VMTS (Pattern Factory) was not part of the Presentation Framework. Although it used the class diagram definition of VMTS, Pattern Factory had a separate data repository and data format to store the patterns. As it has supported UML class diagram components exclusively, this solution was good enough: the modelling space was fixed; there was no need to prepare the application to store various domain-specific language elements. Patterns were stored in separate XML files using a special XML schema. The transformation from this schema to VMTS AGSI schema was done while inserting pattern into models. Obviously, this approach is not enough this case, as our target is to provide a solution which is applicable in all cases with any metamodels without restrictions caused by the fixed data format.

As a metamodelling environment is used, it is recommended to use this modelling environment and its data repository to create and store patterns as well. The simplest solution is to define patterns as general models, but we have to take care of is the right selection of the metamodel, as the pattern can be applied only in a modell with the same metamodell the pattern has. By selecting this way, we have the possibility to store default modell element properties and layout information together with the modell as well. We do not have to extend the modelling environment neither to create nor to store patterns.

It is not enough to create patterns on their own; there is a natural need for the capability of organising them into pattern repositories and attaching some meta information to the patterns as well. This way we can handle a large amount of patterns easily. This could be solved with an external repository like in Pattern Factory, but it is more convenient to use the modelling space with the data store offered by the modelling environment for this purpose.

There are two possible solutions for this: either we can define metamodel-level pattern components, or we can create general, metamodell independent pattern repositories. If we create a meta-pattern element on the metamodell level, we have the possibility to use patterns as components: they can be inserted to the modell as simple elements, and various relations can be established between modell elements and the patterns itself. The main disadvantage of this solution is that it requires modifications on metamodell level to prepare the modelling space for handling patterns. Extending the metamodell with element which do not belong strictly to the modell itself is not acceptable in most cases.

Another possibility for organizing patterns is to create pattern models that are independent from target modell. In this case, we have to create a pattern metamodel, which contains only one element – called MetaPattern – without any relations. The MetaPattern element contains some metainformation about the pattern itself and a reference to the pattern modell. To reach this goal, a Title and a Description attribute with a simple string type is added to the metaelement. Furthermore a TargetModel attribute is created to store the identifier of the modell that contains the domain specific pattern modell itself.

Consequently we have created a metamodel, whose instances contain modell elements. These contained elements represent a reference to the pattern-models themselves. The instance models behave as pattern repositories as they can contain numerous references to real design pattern models.

To be able to apply patterns in target models there was a need for extending the modelling framework with pattern browsing capabilities. Furthermore, it was required to design and implement the logic of building pattern element hierarchy in the target modell by restoring references. Giving the possibility to customize pattern element attributes is a natural need as well.

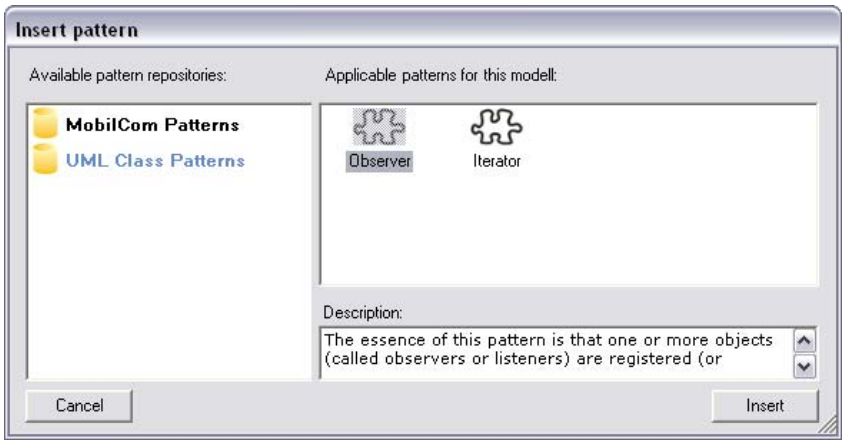

Figure 2
Inserting patterns

By selecting Insert Pattern menu item in the editor, we have the possibility to browse through available pattern repositories and applicable patterns contained by them. Enumerating pattern repository models is simply done by iterating through available models and selecting those which have the same metamodell as the selected Pattern Metamodell (containing the MetaPattern element). The referenced modell of the selected Pattern element decides whether a pattern is applicable in the actual modell or not. Only those patterns are available whose referenced modell has the same metamodell as the target modell.

As some customization facilities during pattern insertion would be welcome, a dialog such as that illustrated in Fig. 3 is offered the user. Here, we have the possibility to change some of the attributes defined in the pattern element. Only element names are offered by default, since there is a need in almost every case to customize them. Additionally, simple attributes whose values are bounded by a '#' character in the pattern modell are also offered for insert-time modification, such that element properties can be adjusted to already existing modell components.

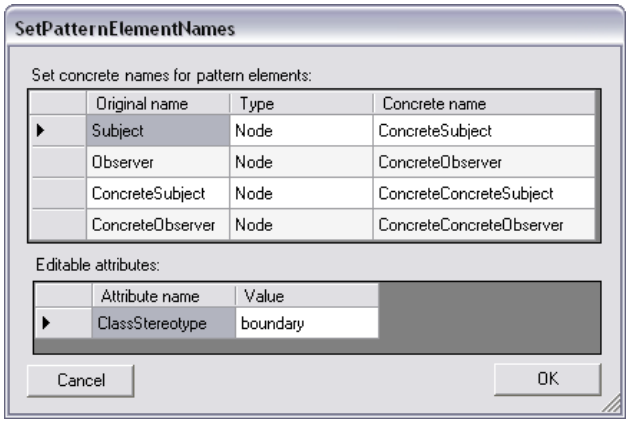

Figure 3
Customizing pattern element attributes

The pattern insertion itself happens in two main phases: In the first phase we create and customize modell elements found in the pattern model, in the second phase we restore relations between these elements in the target modell. Creating modell elements in the target modell means traversing the source pattern model, and instantiating a new modell element for each source element conforming to the metamodell. If the modell element has already been created we can restore visualisation settings and set customized attributes.

AGSI stores all element information using XML documents. Visualization and modell information are separated to InfoXML and PropertyXML documents. Restoring visualization information is applied by overwriting the default InfoXML field of the target modell node with the InfoXML field of the source pattern node. (Visualization information contains sizes, colours, style and other information about the appearance of the element.)

Modell information stored in the PropertyXML can not be copied directly to the target model, because property fields may contain references to other nodes within the modell of the pattern (by containing the identifiers of the referenced node), and these references should be kept in the target modell as well. As new identifiers are generated for the inserted modell elements, the references would be broken. It is straightforward to store source-target modell node identifier pairs. After creating all nodes and all of the source-target pairs are known, we traverse the PropertyXML of the newly created nodes in a recursive manner. Known node-identifiers are replaced in PropertyXML with the value stored in the list mentioned earlier. Afterwards, attribute values in the PropertyXML are restored using the values defined by the form introduced in Fig. 3.

When nodes in the target modell are restored, we can restore relationships between them as well. This step is accomplished similarly: we enumerate edges in

the source pattern modell and create the corresponding metamodel-instances in the target modell. Then we restore modelling and visualisation information by updating PropertyXML and InfoXML fields, finally left and right endpoints of the edges must be replaced with the target modell pair of source modell nodes in order to restore relationships in the target modell properly. For this purpose, the same identifier list can be used as explained in the previous paragraph.

VMTS supports docking elements as well. Docking means that graph nodes can behave as containers of other nodes (See Fig. 4). This requires containment relationship between nodes on modelling level and proper setting of visualization in the InfoXML document. After all nodes and relationships are created and actualized, we can restore the docking information by modifying the InfoXML of the contained and the container node.

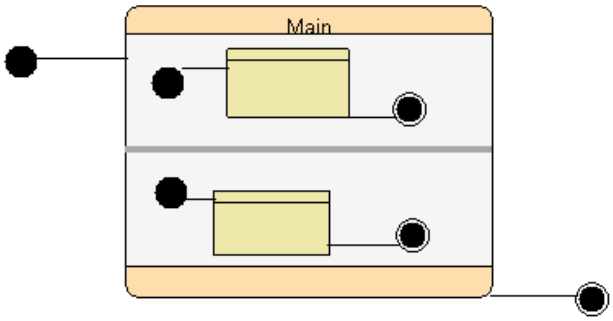

Figure 4
Docking in VMTS

5 Case Study

In this section, a possible application area of patterns is introduced. This area is far away from computer sciences and software modelling for the reason of presenting the flexibility and universality of domain-specific languages and patterns. The chosen topic for modelling is electric circuits, focusing on the application domains of operational amplifiers.

An operational amplifier is a high-gain electronic voltage amplifier with differential inputs and usually a single output. The amplifier has two inputs: an inverting and a non-inverting, only the difference between these inputs is amplified in an ideal case. If there is no feedback between the output and the inputs of the amplifier, the amplifier runs ‘open loop’, and its output value is calculated as

$$V_{out} = (V_{+}-V_{-})*G, \quad (1)$$

where G means the gain of the amplifier. An ideal amplifier has an infinite open-loop gain, infinite bandwidth, as well as infinite input and output impedance, so operational amplifiers are usually used with negative feedback – which means a feedback from the output to the inverting input – for resulting in a finite output voltage value.

The usual circuit symbol of an operational amplifier can be seen on Fig. 5, where V- and V+ mean the inverting and non inverting inputs of the amplifier, Vout is its output , positive and negative power supply are denoted by Vs+ and Vs-

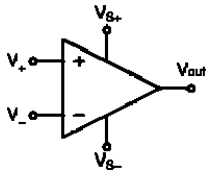

Figure 5
The usual circuit symbol of an operational amplifier

Our basic level schematics should be able to contain some other elements as well to have the possibility to build simple circuits. These elements are resistor, capacitor, power line to connect elements, junction point to connect lines, power supply and ground.

To be able to create models based on these circuit elements the metamodell of the domain is created (Fig. 6).

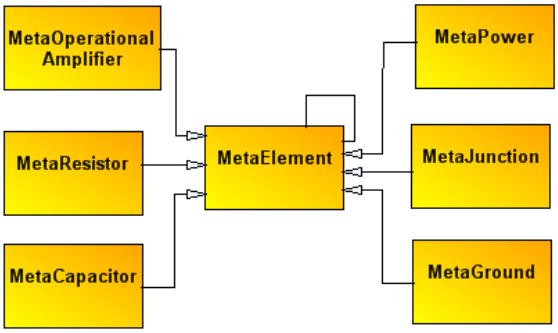

Figure 6
The metamodell of schematic models

The aim of the MetaElement node is serving as an abstract root element for all the other items, so by defining the Connection relationship between these elements, the same connection relationship can be used between all real elements. A Connection has two attributes: LeftNodeConnectionID and RightNodeConnectionID which are used to identify the connection points (in real world: pins) of the elements they are connected to.

As in our simple modell we are using ideal components, we do not need to store any special data neither for an amplifier or ground or junctions. Capacitors, resistors and power supplies have a simple numeric attribute with the name of Value to store resistance, capacity and voltage values.

The next step – after creating the metamodell – is to create a VMTS plug-in for it to have proper visualization for modell elements. This plug-in is responsible for connection snapping to element connection points, and the ability to select a connection point on the elements when connecting a line to them. Presenting how to create such a plug-in is not the aim of this paper, but a detailed description can be found in [10].

The first pattern (model) we create using this metamodell is a Voltage Follower which is presented in Fig. 7.

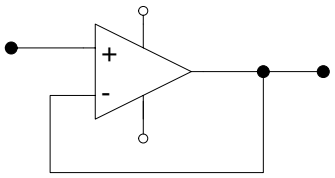

Figure 7
Voltage Follower

This is a really simple pattern, it only provides the same voltage on its output as it receives on its non-inverting input. This is useful as an ideal operational amplifier has an infinite impedance, thus, it does not mean any load for the source system but its output tolerates any load.

The second and third patterns are called Non-Inverting Amplifier and the Inverting Amplifier, respectively. These patterns are illustrated in Fig. 8. The aim of these patterns is to control the gain of the amplifier by adjusting the relation of R1 and R2 resistors. The gain of the Non-Inverting Amplifier can be calculated as

$$1+\frac{R1}{R2}, \tag{2}$$

similarly the Inverting Amplifier has a gain of

$$-\frac{R2}{R1}. \tag{3}$$

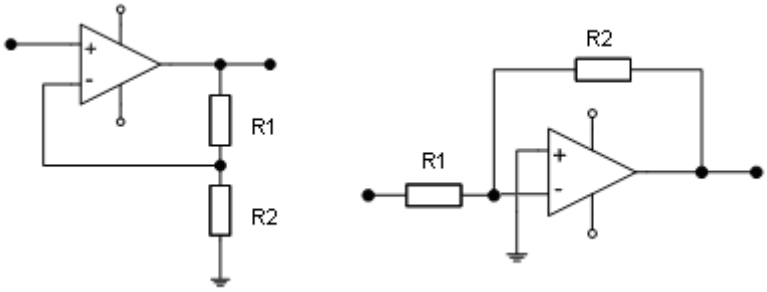

Figure 8
Non-Inverting and Inverting Amplifier

The last two patterns we are going to introduce are the High-Pass Filter and the Low-Pass Filter presented in Fig. 9.

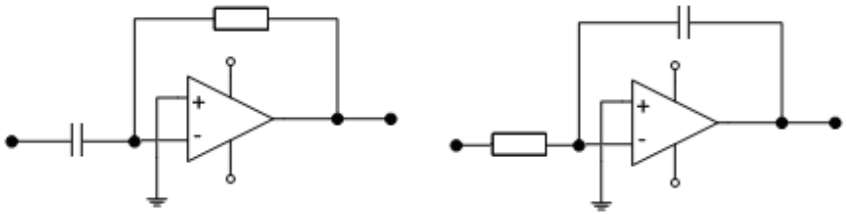

Figure 9
High-Pass and Low-Pass Filter

The difference between these filters and the previously introduced Inverting and Non-Inverting Amplifiers is that either of the resistors is replaced with a capacitor. A capacitor has a frequency-sensitive impedance, by improving the frequency of the applied AC, the impedance of the capacitor will become smaller. That is why these patterns are called filters, as they have a frequency-dependent gain.

After creating a Pattern modell for storing references to these modell instances we have the possibility to apply these patterns in other schematic models as well. A simple real world application of some of these patterns is the Buffered Inverting Amplifier which is widely used in signal-processing applications such as audio and video processing. It consists of an Inverting Amplifier preceded and followed by Voltage Followers, thus, it can be created simply by inserting these patterns and connecting them with two power lines. The aim of Voltage Followers is to avoid interaction between the source system and the Inverting Amplifier, and between the Inverting Amplifier and the target system. The final layout of this application can be seen in Fig. 10.

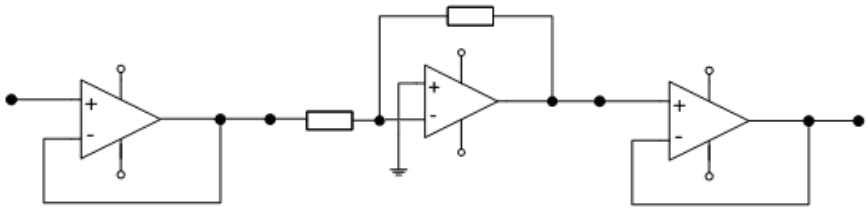

Figure 10
Buffered inverting amplifier

Conclusions

In this paper we have introduced the strength of using patterns in Domain-Specific Languages and gave a powerful solution for creating, organizing and applying patterns in an existing, widely known metamodelling environment (VMTS). We described how to traverse and rebuild patterns in other models keeping modell hierarchy, modell properties and visualization at the same time. Via the case study, we presented the great perspectives of this solution outside the software modelling world as well.

Future work includes organizing patterns into standalone components realizing Component Oriented Modelling. This would require making modifications in metalevel models, or it would be more elegant to realize model-inheritance and giving component elements as aspects to the metamodell.

Acknowledgement

The found of ‘Mobile Innovation Centre’ has supported, in part, the activities described in this paper.

References

[1] UML - www.uml.org

[2] Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional Computing Series

[3] Visual Modeling and Transformation System, http://vmts.aut.bme.hu

[4] Mezei G., Levendovszky T., Charaf H.: A Presentation Framework for Metamodelling Environments, 4th Workshop in Software Model Engineering, October, 2005, Montego Bay, Jamaica

[5] Rational XDE, http://www-128.ibm.com/developerworks/rational/products/xde/

[6] VIATRA 2, http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/subprojects/VIATRA2/index.html

[7] Herzner W., Csertán Gy., Balogh A.: Design Patterns for Domain-specific Application Modelling 2006, ERCIM / DECOS Workshop on Dependable Embedded Systems, 2006

[8] Borland Together, www.borland.com/together

[9] Rational Software Architect, http://www-306.ibm.com/software/awdtools/architect/swarchitect/index.html

[10] Mezei G., Levendovszky T., Charaf H.: Automatized Concrete Syntax Definition for Domain Specific Languages, Proc. of the 7th International Conference on Technical Informatics, 2006