e-Informatica Software Engineering Journal, Volume 3, Issue 1, 2009

Tool Based Support of the Pattern Instance Creation

Ľubomír Majtás[*]

[*] *Faculty of Informatics and Information Technologies, Institute of Informatics and Software Engineering, Slovak University of Technology*

majtas@fiit.stuba.sk

Abstract

Patterns introduce very useful way of improving the quality of the software development process. Nowadays modeling tools and techniques provide some kind of support for modeling with pattern instances, but these are often based on manual pattern creation and connection to the rest of the model. Our approach presents the support of pattern instance creation on the model level in semi automatic way that simplifies the whole process. The main idea of this approach is that the developer should assign the domain dependent parts of pattern and specify the requirements over the pattern variants. The rest of the pattern instance is to be created by the machine.

1. Introduction

Pattern introduction [1] had large asset for more areas. Probably the most familiar pattern's fulfillment in the software engineering was introduced by the work of GoF [12], where the authors identified and in detail described 23 design patterns. Their description of each pattern contains the verbal description of its main idea, the example of its appropriate usage (including the source codes), the description of solution it offers and discussion about its alternatives and consequences of its usage. The main part of description is presentation of the pattern model according to the OMT/UML diagrams. The authors provided patterns' explanations by examples and textual description so the catalog means a useful knowledge base for software professionals. On the other hand they did not try to present any "computer friendly" knowledge that could be basis for automation of typical pattern processes which are:

– Creating of pattern custom instances,
– Validating existing pattern instances,
– Identification of pattern instances in existing codes.

To solve this drawback, there were presented many other works that extend the original catalog by the different models that are trying to capture the core structure of patterns, e.g. [11], [13], [7].

In this paper we would like to introduce the approach that would support developers in their work with pattern instances. Nowadays modeling tools and techniques provide some kind of support for modeling with pattern instances, but these are often based on manual pattern instance creation and connection to the rest of the model. Our approach presents the support of pattern instance creation at the model level in semi automatic way that simplifies the whole process. The core idea of the approach is that the developer should assign the domain dependent parts of pattern and specify the requirements over the pattern variants. The rest of the pattern instance should be generated automatically. In this paper we will analyze the processes taking place while creating the pattern instance. We will identify the places where can be this process

automated. Finally we will provide our approach of automation in a way that will support but not limit the developers.

2. Process of the Pattern Instance Creation

Process of the pattern instance creation means the application of the solution offered by the pattern to the environment of the developed software system. The inputs of this process are the actual environment of the developed software and the general description of the pattern. As the output we consider modified software system extended by correctly created instance of the pattern.

We distinguish two activities that are necessary to follow out while creating the pattern instance [15]: abstract and general pattern instance needs to be concretized and specialized. At the first moment both activities seem to be similar, but it is not so. Each one moves the first idea of the pattern application to the final instance, while it is necessary to follow up both, to be able to declare the pattern instance as the correct one. Differences between these activities are presented in the Figure 1, where the degrees of generality and abstraction are being presented in two dimensional space (degree of generality horizontally, degree of abstraction vertically).

The created instance is becoming more concrete when it contains more building blocks creating the correct instance. To the beginning abstract idea of the pattern application there are subsequently being added classes, their attributes, methods and relations until instance becomes complete. Specialization of the instance means the movement of the general pattern description to the context of the developed system. The specialization follows such modifications of the pattern instance that make the instance domain specific and subsequently specific for the current software system. As the examples of the specialization steps we can consider definitions of roles' participants count, naming of the participants or creating the relations between participants according to the domain.

In our approach we look for possibilities for automation of the pattern instance creation. We see higher potential in the process of concretization than in process of specialization. The specialization is based on the ability of developer to move the pattern to the particular target software environment. It can be seen as a domain based pattern description, what we consider as almost impossible to be performed by the machine. There can be found only minimal space for automation of this process. On the other hand, there is a potential for tool based support of concretization process. When the pattern instance is correctly specialized, its concretization is often based more on pattern structure description than on developer's skills. It means that there is a space for automation of this process.

2.1. Pattern Roles, Domain Dependency

Patterns are often being described as a collection of cooperating roles. These roles can be often divided into two groups: roles dealing with the domain of the created software system (domain roles) and roles performing the pattern's infrastructure (infrastructure roles). 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 frequently between the pattern instances. Their purpose is to glue the domain roles together to be able to perform desired common functionality.

One of the main contributions of the whole pattern approach is that it allows thinking at the higher level of abstraction. Developers do not have to always keep in mind all details about the solution, they can work with the pattern instance as with single unit hiding unnecessary complexity. When the developer thinks about applying the pattern to the project, first thing he needs to decide is how to connect pattern instance to the context of the software. He does so be specifying the domain roles' participants. The other issues are often second-rate at that moment. Table 1 describes selected patterns and

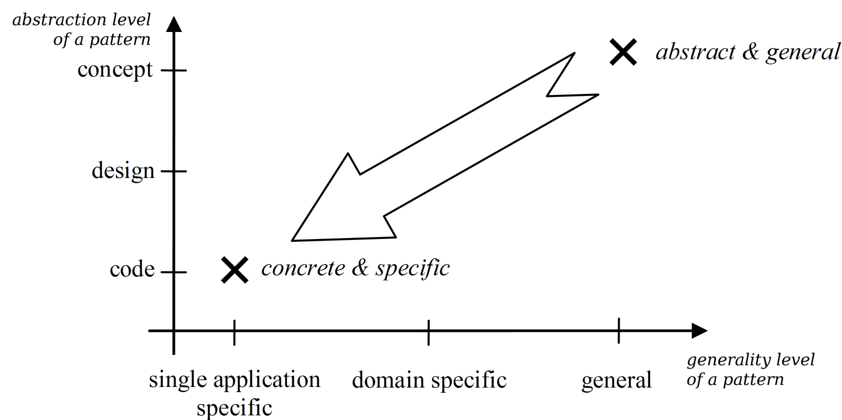

Figure 1. Two dimensional space of generality and abstraction [15]

Table 1. Specialization of the domain dependent pattern roles

Pattern	Domain dependent roles	Description
Composite	Leaf and its Operations	Leafs and their operations provide all domain dependent functionality. Everything else is just infrastructure allowing the hierarchical access to the leaf instances.
Chain of Responsibility	HandleRequest, Ancestor	The domain dependent is the business logic processing the event and the ancestor to which should be the unprocessable event passed.
Decorator	Concrete Component, Concrete Decorator	The domain dependent are the Concrete Component (which often exists before Decorator pattern application) and functionality of Concrete Decorator participants that provide extended functionality to the Concrete Component.
Flyweight	Concrete Flyweight	Concrete Flyweight provides all domain dependent functionality. The rest is infrastructure for storing instances in memory and providing access to them.
Proxy	Real Subject, Proxy	The domain dependent is the Real Subject (which often exists before Proxy pattern application) and functionality of Proxy participants that provide access to the Real Subject.

their roles from the perspective of the domain dependency.

2.2. Pattern Variability

As the pattern variability we understand the possibility to provide patterns functionality in slightly different ways. Each variant of the pattern has its own pros and cons and therefore the decision which variant to select does not need to be easy. Selecting the proper variant is part of pattern instantiation process, when we are customizing instance according our needs. By the variability we do not understand definition of participants playing defined roles.

General understanding of the patterns does not always satisfy the ideas of their authors. Many developers understand design patterns only as constant templates with strictly specified purpose of each class, attribute or method. However the original idea was to discuss a problem and offer a solution. Examples provided with pattern description were never meant as the only best solutions. Their purpose was to

Table 2. Examples of possible variants of pattern instances

Pattern	Variability description
Abstract Factory	Class structures do not differ much. Majority of variants are dealing with realizations of ConreteFactories: they can be implemented normally or as Singletons, they can employ Factory Method or Prototype patterns.
Adapter	There are more different variations how can Target, Adapter and Adaptee communicate together.
Bridge	Possible variants: Omitting the Implementer interface in case of the only one ConcreteImplementor.
Chain of responsibility	References to the successor can be maintained commonly in Handler or custom for each ConcreteHandler. Handler can be a class with default forwarding functionality or just an interface.
Flyweight	The Flyweight interface can be omitted.

provide hint, to show one of many possible ways of idea realization.

When we look closer on the patterns from the GoF catalog, we can distinguish differences of examples generality between patterns. On the one side stands Singleton. It is very simple with accurate example that is not keeping much space for different variations while it prescribes all desired functionality. On the other side we can see patterns such as Memento. Their examples are very general; they do not provide expected functionality but only briefly draft the solutions. Their concrete instances can be far different from the presented examples. In the middle of these extremes stands majority of the patterns. Their examples are able to provide desired functionality while they are keeping a space for their customization. Typical representatives are Composite, Observer or Decorator.

From the perspective of tool based support of instantiation, it is very difficult to create support for instantiation of pattern with very general examples. It would be very difficult (if it is even possible) to automatically instantiate fully functional Memento that would fit to the rest of the system. On the other hand there are minimal difficulties for pattern with strictly defined structure keeping minimal space for variability. Pattern such as Singleton can be automatically instantiated with minimal efforts. The simple template based instantiation would be sufficient. For the majority of patterns the simple template based approach cannot cover all known variability, such approach cannot be considered as sufficient. In this case we need to employ approach that is based on templates that are created according the user needs. In Table 2 we present examples of possible variabilities of selected GoF design patterns.

2.3. Pattern Instantiation Support in CASE Tools

Many existing CASE tools are trying to provide some kind of support in pattern instantiation process. The level of such support differs. Often they allow inserting of example pattern instances to the model. The others can run wizards through which developer can specify the participant count of the selected roles and specify the name for each one. In general the support is based on single template, where the developer can or cannot specify the participants before the creation of the instance. Advanced CASE tools are tacking the information about pattern occurrence (often by UML Collaboration element). This helps the further developers to identify the pattern with minimal effort and thereby it reduces the risk of instance damage that can happen by improper modifications in later project phases (e.g. maintenance). However, these tools do not try to automate the process pattern instantiation – participants of all roles need to be specified by developers. Alike the support for other kinds of customizations is often omitted; it is left to developers' knowledge and experience to modify the instance according their needs.

Employing the automation of pattern instantiation in CASE tools can lead to the following benefits:

- Developers do not need to perform typical modifications manually. By minimizing the effort that needs the developer to perform to create pattern instance or by giving him possibility to select the proper pattern variant they can save time and avoid mistakes, so the instantiation process becomes more effective.
- Developers do not need to know all pattern complexity or inner structure. They can focus on the domain dependent context of the pattern; they "do not need to care" about the rest. This can help the inexperienced developers with pattern application, and in this way support them to utilize patterns in their everyday work.
- Developers can be informed about possible variants. Sometimes developers do not have to know about existence of different pattern variant. When they are informed immediately about more possibilities they can choose most proper variant without former knowledge about it. Developers are able to get best from the pattern application.

3. Our Approach of the Tool Based Support

In the previous sections we have described why should we consider the tool based support for the pattern instance creation and where are the spots for the automation. In this section we will describe how can be such automation provided. We focus on two different ways of support. The first one is dealing with the process of instance creation, it lets the developer to define domain based participants and automatically supplement infrastructure participants to form a valid instance. The second one provides support for pattern variability; it informs the developer about possible variants of the pattern, asks for desired ones and automatically reasons the valid configuration according the developer's choice. The result of this step is the role based model of the proper pattern configuration, which comes as an input for the first mentioned support.

3.1. Pattern Inner Structure Description

To be able to provide machine based pattern processing, we require precise models of the patterns' inner structures. We are using custom role based models which are defining the collaborations between the roles that perform the predefined functionality. As the roles we do not consider only ones typically played by classes but also ones which participants are attributes or methods. Only the definitions of the roles do not capture whole pattern inner structure and therefore cannot be used as the blueprint for the machine based pattern instance creation. The very important parts of the pattern structure are the definitions of the inner structure constraints. These constraints associate roles that are somehow linked together. Example of such constraints can be clearly seen in the Abstract Factory pattern where the roles createProduct() of Abstract Factory and Abstract Product are linked together. It means that there has to be the same count of participants of this role where each participant of the createProduct() role is responsible for creation of the appropriate participant of the Abstract Product. The exact definitions of constraints are very important in the process of machine based pattern instantiation while they help to specify the count of all participants and set the proper links between them. We have identified and capture in our models the following relationships which are bases of constraints:

1. Inheritance – inheritance between classes,
2. Association – associations between classes,
3. Overriding – in case of inheritance where the one method role overrides the other method role,
4. Method delegation – one role invokes other role to delegate the functionality,
5. Instance creation – role creates the instances of other role,
6. Class linked with its members – role which participants are regularly classes and can be played by more than one participant needs to be explicitly linked with its method and attribute roles.

Some roles are part of definition of more than one constraint. For example in the Abstract Factory pattern the count of participant of role Concrete Product is dependent on count of participants of roles Concrete Factory and Abstract Product. We say that the dimension [13] of the role Concrete Factory is two because it is part of two different constraints.

3.2. Algorithm of the Pattern Instance Creation

Our process of pattern instance creation is based on supplementing the incomplete pattern instance defined by developer. As the inputs the algorithm requires proper role based pattern model (according to the previous section) and the partially created pattern instance containing some participants of the domain role. The algorithm progressively adds the missing participant to comply the pattern's inner structure description and all constraints of the pattern. The output of the algorithm is the model of pattern instance containing all participants that are meeting all constraints.

The algorithm stores information about all participants which are part of the pattern instance at the moment. Moreover it stores information about instances of all constraints connecting the linked corresponding participants. Some roles are present in more constraints (their dimension is more than one) what causes that instances of constraints overlap over the participants of such roles. Therefore the algorithm stores the information about instances of constraints in the n-dimensional structure where n is the maximum dimension of all pattern roles (the GoF pattern do not have roles with dimension more than 2). We call this structure Participant Constraint Matrix (PCM). Each dimension of this matrix corresponds to one pattern constraint. Lines in this dimension represent the instances of constraints. These instances of constraints link corresponding participants according to the constraint. Lines cross in the places of more dimensional participants. Examples of partially filled Participant Constraint Matrix are depicted in the Figures 2, 3 and 4.

In the following section we describe the steps of the algorithm creating the pattern instance. We will describe the algorithm also on example, each step will contain example of execution results of this step. In our example we will create the instance of the pattern Composite. As the input we get the incomplete instance containing only partial definitions of two participants of the domain role Leaf: Leaf1 containing the Operation1() and Leaf2 containing the Operation2(). The algorithm will create the correct instance according these inputs. The algorithm takes the following steps:

1. Add participants of non constrained roles. Create the participants of the roles that are not concerned in any constraint. For each role create exactly one participant and name it as the role. Create the links that are related to the new participants.
   Example: Add the participants of roles Component, Composite, Composite's childs and links between them: generalization and association.
2. *Create the empty Participant Constraint Matrix.* Create the empty PCM according to the definition of pattern constraints.
3. *Initialize the PCM according to the current of the pattern instance.* Fill the PCM with already created the participants.
   Example: Fill the PCM as depicted in the Figure 2.
4. *while (the PCM contains empty fields)*
   a) *Add participant to fill one empty field of PCM.* Select one empty field of the PCM and create the participant that will fit to this field. When selecting the empty fields, start with class participants and continue with association and method participants. Prefer empty fields with lower dimension.
      Example: In the first iteration create the participant of role Component's Operation().
   b) *Add information related to the added participant.* Fill the information about the new participant and add connections

			Leaf members constraint	
			<<Leaf>> Leaf1	<<Leaf>> Leaf2
Operation() overriding constraint	<<Component's operation>>	<<Composite's operation>>	<<Leaf's operation>> Leaf1's Operation1()	<<Leaf's operation>>
	<<Component's operation>>	<<Composite's operation>>	<<Leaf's operation>>	<<Leaf's operation>> Leaf2's Operation2()

Figure 2. Initial Pattern Constraint Matrix for incomplete instance of pattern Composite

			Leaf members constraint	
			<<Leaf>> Leaf1	<<Leaf>> Leaf2
Operation() overriding constraint	<<Component's operation>> Component's Operation1()	<<Composite's operation>>	<<Leaf's operation>> Leaf1's Operation1()	<<Leaf's operation>>
	<<Component's operation>>	<<Composite's operation>>	<<Leaf's operation>>	<<Leaf's operation>> Leaf2's Operation2()

Figure 3. Extended of Pattern Constraint Matrix after adding Component's Operation()

			Leaf members constraint	
			<<Leaf>> Leaf1	<<Leaf>> Leaf2
Operation() overriding constraint	<<Component's operation>> Component's Operation1()	<<Composite's operation>> Composite's Operation1()	<<Leaf's operation>> Leaf1's Operation1()	<<Leaf's operation>> Leaf2's Operation1()
	<<Component's operation>> Components Operation2()	<<Composite's operation>> Composite's Operation2()	<<Leaf's operation>> Leaf1's Operation2()	<<Leaf's operation>> Leaf2's Operation2()

Figure 4. Pattern Constraint Matrix for complete instance of pattern Composite

with the other participants that are related to the new one, e.g. generalization, overriding, association, delegation, etc.
Example: Connect the participant with Leaf's Operation1() with overriding relationship. Name the participant Operation1() because the overriding relationship needs the same name of the linked participants. Extended PCM by this step is depicted in the Figure 3.

After successful execution of the algorithm the pattern model of the instance is created. It complies with all rules and constraints coming from the pattern description. The created model keeps information about the role that participants play and therefore can be used for further source code generation of the pattern instance. PCM for the complete pattern instance is depicted in the Figure 4.

3.3. Pattern Variability

As mentioned in previous sections patterns are not simple units with the only one valid template. Most of them are highly customizable allowing changes in their example templates in many different ways. In this section we describe the approach of employing the variability to the instantiation process. The result of this step is role based model describing the customized pattern template that stands as an input for previous algorithm.

3.3.1. Variability Modeling

To capture possible variability, we are employing feature modeling technique that was originally designed for the product-line engineering [6]. It is important for capturing and managing commonalities and variabilities in product lines throughout all stages of product-line engineering. In early stages it is used for scoping of product line (i.e. deciding which features should be supported by a product line). In product-line design, the variation captured from feature models are mapped to product-line architecture common for all parallel product lines. In the product development, feature models can drive requirements elicitation and analysis, help in estimating development cost and effort, and provide a basis for automated product configuration. Feature models are also important in generative software development which is trying to automate application engineering based on system families.

In our approach we use the feature models to capture possible variants of the patterns. The model depicted in the Figure 5 presents selected variabilities of the Composite pattern. It says for example that participant of the role Component can be either the interface or the abstract class or that the processing method of the Composite's children role can be omitted, present only in the Composite or in the whole structure. The feature model also presents relations between the features. For example it is not possible to omit these processing methods when the Composite's children is private attribute. Such configuration would disable the whole pattern's functionality because the structure would become unmodifiable.

The presented model is considered only as an example. It does not cover all the variabilities such as the way of Composite's children collection realization.

3.3.2. Configuration Reasoning

When creating a pattern instance, developer needs to specify his requirements dealing the variability. The target is to specify whole feature selection (also called product configuration), which is a group of desired functional capabilities that constitute a complete configuration of an application and adhere to the constraints specified in the feature model. We do not want to force the user to provide information about each feature whether he wishes to employ it or not. We give him a chance to specify which features he wishes to employ and the rest of the configuration is set by the tool. The final configuration has to fulfill all constraints defined by the feature model, so if the developer's requirements do not meet these constraints or do not allow creation of valid configuration, the developer has to be asked to change his preferences.

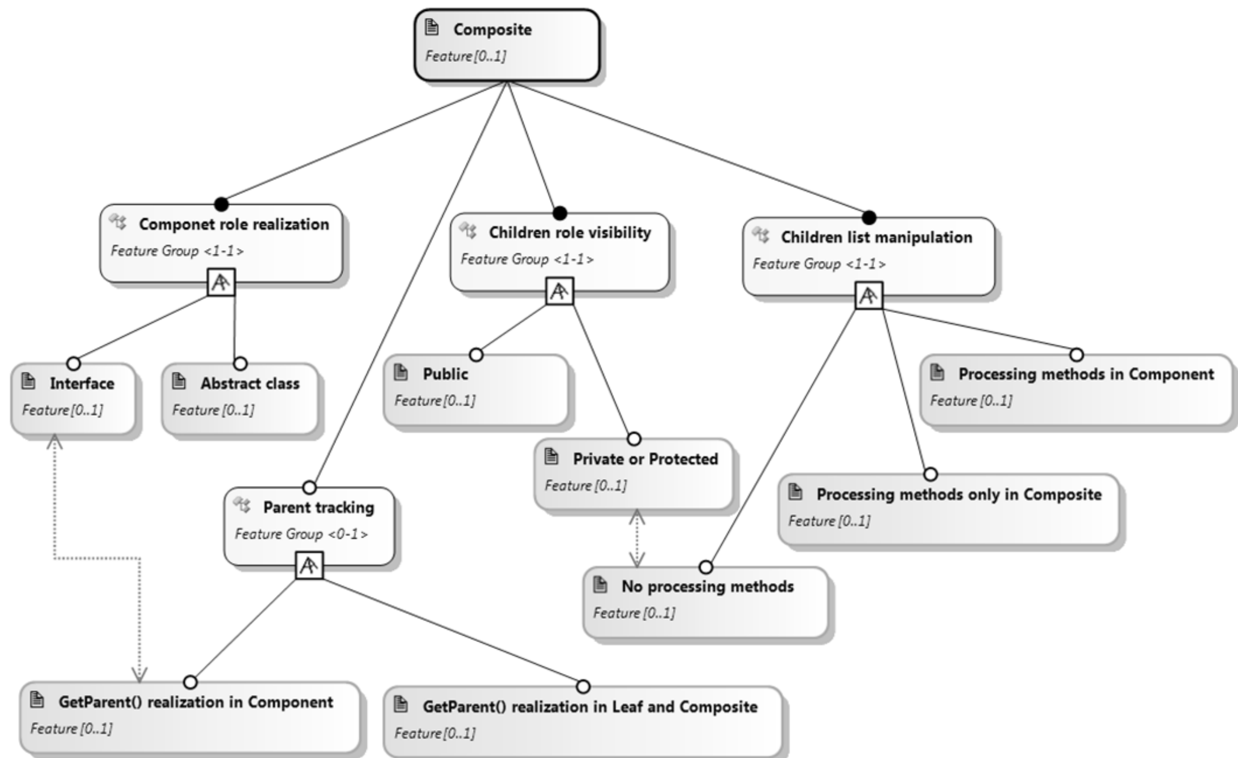

Figure 5. Feature model example capturing the Composite pattern variability

To reason a valid configuration we transform the feature model and partial configuration based on developers preferences to Constraint Specification Problem (CSP) environment and apply existing CSP solver to reason final configuration or to notify us about impossibility to finish this task. To create the CSP from a feature model we apply transformation rules specified in [2]. As the CSP solver we chose Choco CSP [4] which is an open source Java based software.

3.3.3. Final Model

When the configuration is set up, we are able to provide concrete role based model of the pattern instance. The variants represented in the feature model have several possible impacts to the final instance, while they can:

- Prescribe the role based model
  - Specify the occurrence of the role, whether participants of the role should be part of pattern instance or not.
  - Specify the position of the role. For example specify, whether the operation should be present in parent class or only in child classes.
- Define the implementation aspects relevant for code generation
  - Specify the form of participants' realizations. For example they can specify whether class role would be played by class or interface or whether list will be realized as array, linked list, map or something else.
  - Specify the participants' visibilities: Public/Private/Protected.

The definition of the roles occurrence or positions prescribes the output role based model. It is provided by reduction of the general pattern template containing all variability roles. This template contains all roles including the conditions when should be these role applied (which feature needs to be part of the configuration to activate the variability role). The Figure 6 contains such template for the Composite pattern according the features presented in the Figure 5. The variability roles are filled grey and the activating features are placed next to them as a bold text. According the current configuration, the variability roles with features that are not contained in the configura-

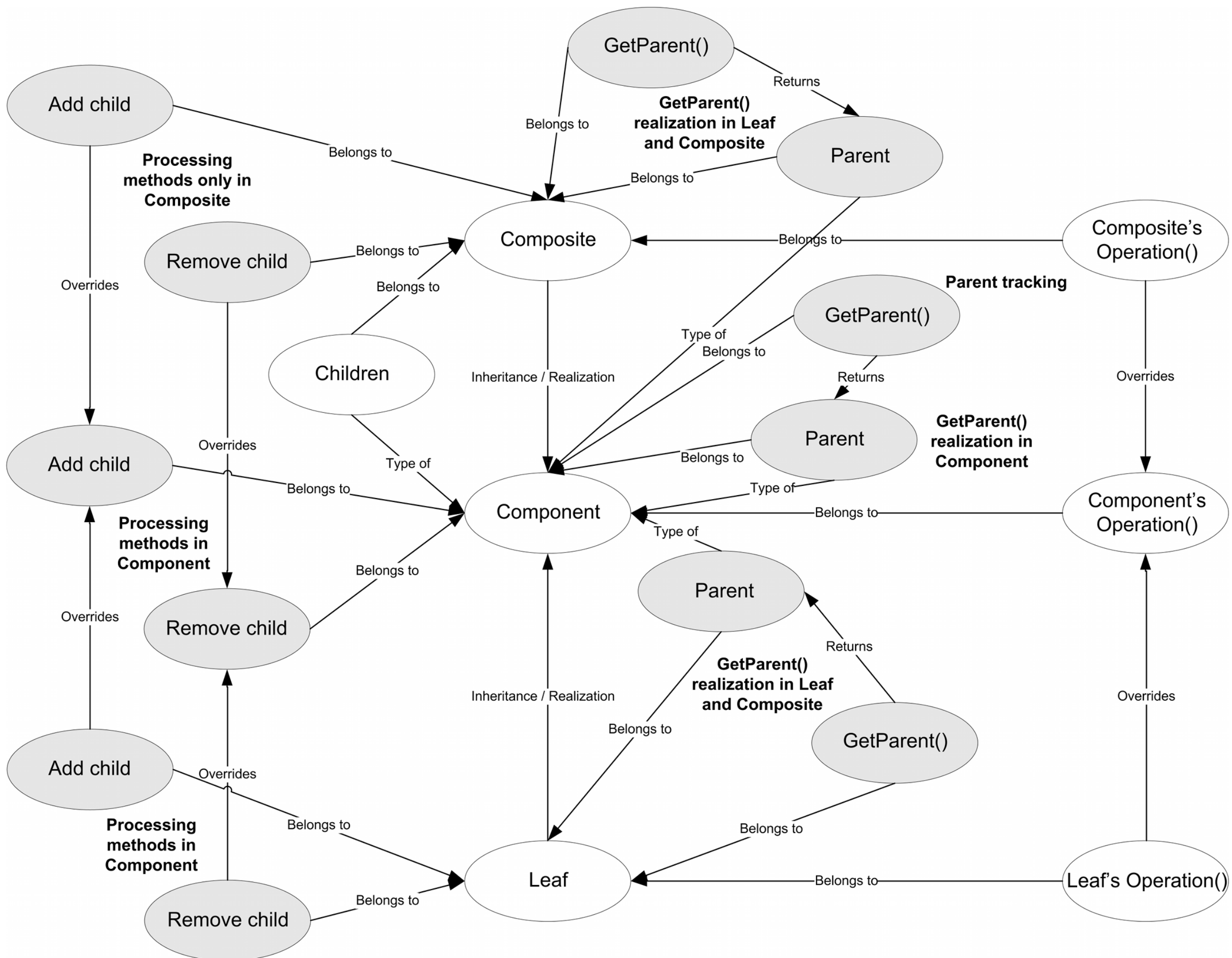

Figure 6. Role based template of the Composite pattern extended by variability roles

tion will be omitted, the rest will form the final role based model according the input configuration.

As an example, when we take the Composite's feature model from the Figure 5 and select the following variants: Parent tracking – GetParent() realization in Component and the Children list manipulation – No processing methods. The final role base model representing the configuration received from the previous inputs is depicted in the Figure 7.

3.4. Overall Instantiation Process

Presented approaches form a complex process of computer aided pattern instantiation deliberating the pattern variability. It consists of the following steps:

1. Inquire the user about pattern he wishes to instantiate;
2. Inquire the desired domain participants;
3. Inquire the desired variants of the pattern;
4. Reason pattern configuration;
5. Create the role based model for the reasoned configuration;
6. Supplement all missing participants from the incomplete user specification according the actual role based model;
7. Create the instance of the pattern.

In general the user is inquired for the domain dependent information and customizations while the tool creates the pattern instance satisfying the user needs. The architecture scheme of the overall approach is sketched in the Figure 8. The general input for the entire process is role based pattern model containing all variability roles. Variabil-

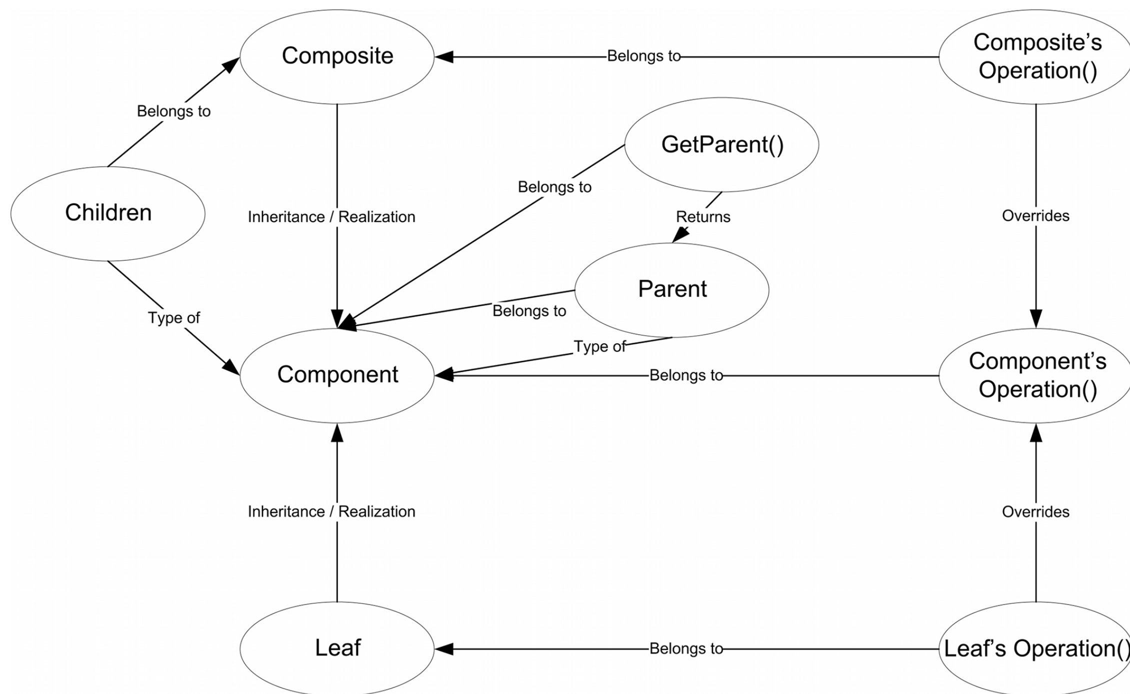

Figure 7. Example of output role based model according the custom configuration

ity support module reduces this model into the concrete role based model for the custom pattern configuration inferred from developer's variants selection. Pattern instance creation module takes this model and according to it and developer's specification of domain dependent participants creates the final pattern instance. This final instance reflects the developer's variants selection and pattern domain specialization.

3.5. Realization

The presented approach was partially implemented and verified. It was realized as the plug-in of the Rational Software Modeler which is based on open source platform Eclipse. The Figure 9 contains screenshots of the model before and after the execution of the overall pattern instantiation process. As the inputs for this scenario we have used the inputs of the examples presented in former sections.

4. Related Work

Different approaches of automating the pattern utilization in software projects were introduced by the other authors. Ó Cinnéide et. al. [5] have presented a methodology for the creation of behavior-preserving design pattern transformations and applied this methodology to GoF design patterns. The methodology is taking place in refactoring process when it provides descriptions of transformations to modify the spots for pattern instance placement (so called precursors) by the application of so called micropatterns to the final pattern instances. While Ó Cinnéide's approach is supposed to guide the developers to pattern employment in the phase of refactoring (based on source code analysis), Briand et. al. [3] are trying identify the spots for pattern instance in design phase (based on UML model analysis). They provide semi-automatic suggestion mechanism based on decision tree combining evaluation the automatic detection rules with user queries.

There exist several approaches introducing their own tool based support for the pattern instantiation. El Boussaidi et. al. [10] present model transformations based on Eclipse EMF and JRule framework. Wang et. al. [16] provide similar functionality by XSLT based transformations of the models stored in XMI-Light format. Both approaches can be considered as

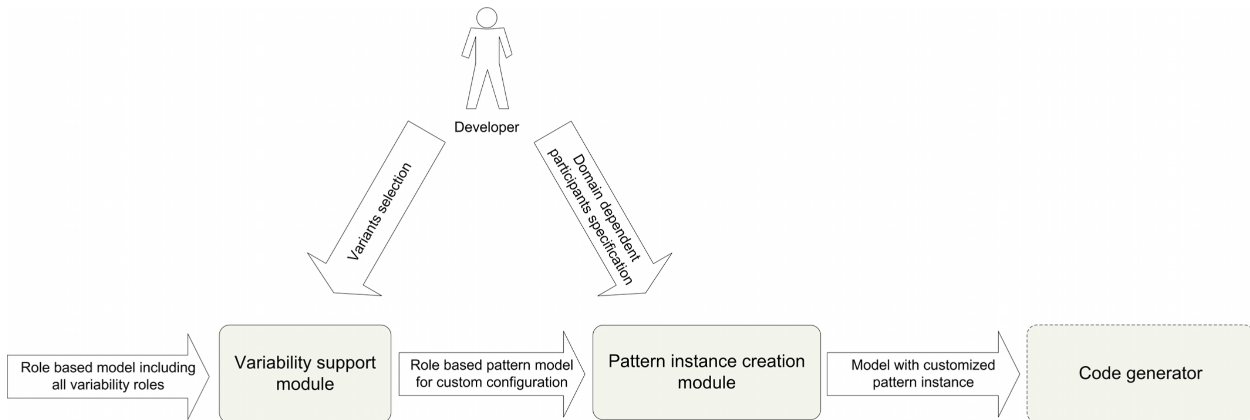

Figure 8. Architecture overview of the overall approach

the single template driven while they are focusing most on the transformation process and do not set a space for the pattern customizations. More advanced method was introduced by Mapelsden et. al [14]. Their approach supports instance configuration by specifying the role participants including those playing more dimensional roles. All of these approaches are based on strict forward participants generation – participants of all roles are created according the single template. Our approach accentuates on collaboration between the developer and the CASE tool. We do not intent to create all pattern participants. We let the developer define the ones he needs and subsequently we infer and create the rest ones to form a valid instance. We do not force the developer to our solution, we let the template and the final instance be customized according the developers' needs.

All the former approaches were focusing on the creation of pattern instances. The ones presented by Dong et. al. presume the presence of the pattern instances in the model. They are providing the support for the evolution of the existing pattern instances resulting from the application changes. The first one [8] implementation employs QVT based model transformations, the other one [9] does the some by the XSLT transformations over the model stored as XMI. However, both are working with the single configuration pattern template allowing only the changes in presence of hot spots participants. Other possible variabilities are omitted.

We have not found any approach regarding the feature modeling application in pattern instantiation area. The feature models were successfully employed in other areas, for example in automation attempt of enterprise application configuration presented by White et. al. [17].

5. Conclusion and the Future Work

In this paper we have presented our approach dealing with a tool based support for pattern instance creation. Our key concept was to create such methodology that would help the developers with application of the pattern solution to their software but allow them to customize the pattern according their needs. We were trying to handle two different courses while more generative parts often mean less space for customization and vice versa. We believe that our approach balances these opposing courses into final solution in a way that forms useful a tool for developers interested in pattern employment.

In the future we would like to extent the created pattern instance model with behavioral information. The correctly created instance would be represented class diagram together with sequence diagram. The main building blocks of such behavioral model will be method invocation and delegation, instance creation together with structural blocks such as condition or iteration over collection. Also we would like to prepare definitions of more GoF design pattern to evaluate the algorithm on the larger scale.

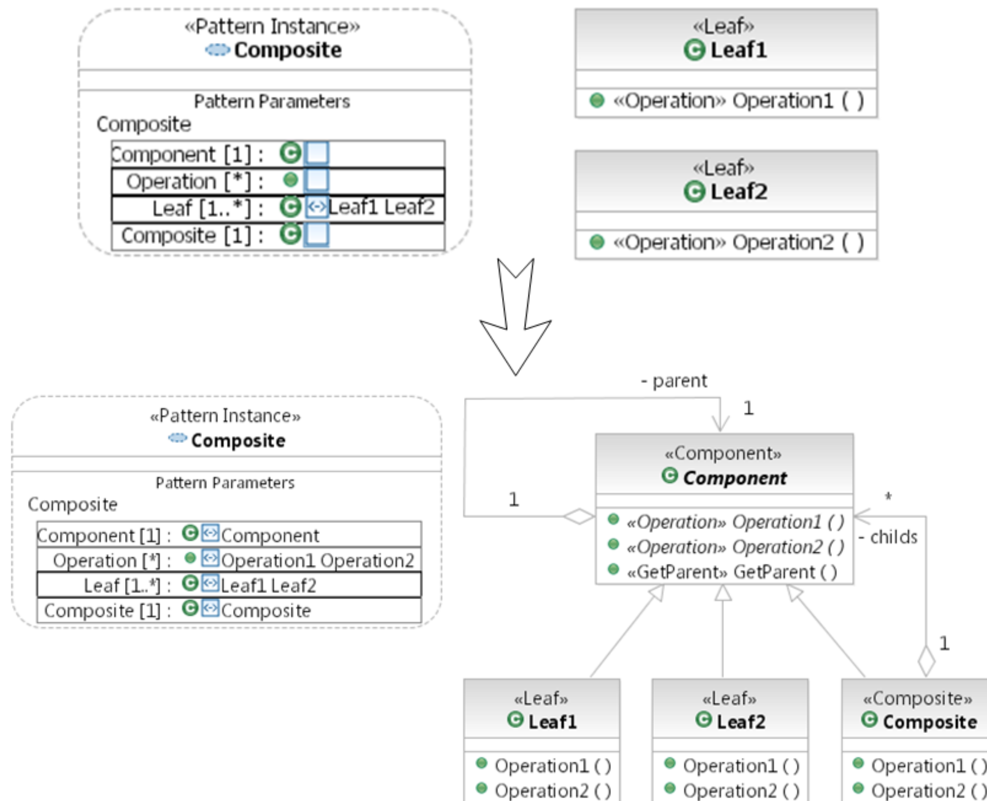

Figure 9. Screenshots of models before and after the overall process execution

We are thinking about other patterns that can be input for the algorithm. It could be applicable on all patterns that are at the design level of abstraction and their inner structure can be described by relation based constraints. Candidates for such patterns are for example the J2EE design patterns.

References

[1] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, August 1977.

[2] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In *LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*. Springer, 2005.

[3] L. C. Briand, Y. Labiche, and A. Sauve. Guiding the application of design patterns based on UML models. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 234–243, Washington, DC, USA, 2006. IEEE Computer Society.

[4] Choco constraint programming system. http://choco.sourceforge.net/.

[5] M. Ó Cinnéide and P. Nixon. Automated software evolution towards design patterns. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 162–165, Vienna, Austria, 2001. ACM.

[6] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[7] J. Dietrich and C. Elgar. A formal description of design patterns using owl. In *ASWEC '05: Proceedings of the 2005 Australian conference on Software Engineering*, pages 243–250, Washington, DC, USA, 2005. IEEE Computer Society.

[8] J. Dong and S. Yang. Qvt based model transformation for design pattern evolutions. In *IMSA '06 : Proceedings of the 10th IASTED international conference on Internet and multimedia systems and applications*, pages 16–22, 2006.

[9] J. Dong, S. Yang, and K. Zhang. 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*, pages 80–92, Washington, DC, USA, 2006. IEEE Computer Society.

[10] G. El Boussaidi and H. Mili. A model-driven framework for representing and applying design patterns. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 97–100, Washington, DC, USA, 2007. IEEE Computer Society.

[11] R. B. France, D.-K. Kim, S. Ghosh, and E. Song. A UML-based pattern specification technique. *IEEE Trans. Softw. Eng.*, 30(3):193–206, 2004.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[13] J. K. H. Mak, C. S. T. Choy, and D. P. K. Lun. Precise modeling of design patterns in UML. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 252–261, Washington, DC, USA, 2004. IEEE Computer Society.

[14] D. Mapelsden, J. Hosking, and J. Grundy. Design pattern modelling and instantiation using dpml. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 3–11, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[15] M. Smolárová, P. Návrat, and M. Bieliková. A technique for modelling design patterns. In *JCKBSE '98: Proceedings of the Knowledge-Based Software Engineering*, pages 89–97. IOS Press, 1998.

[16] X.-B. Wang, Q.-Y. Wu, H.-M. Wang, and D.-X. Shi. 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*, page 24, Washington, DC, USA, 2007. IEEE Computer Society.

[17] J. White, D. C. Schmidt, K. Czarnecki, C. Wienands, G. Lenz, E. Wuchner, and L. Fiege. Automated model-based configuration of enterprise java applications. In *EDOC '07: Proceedings of the 11th IEEE International Enterprise Distributed Object Computing Conference*, page 301, Washington, DC, USA, 2007. IEEE Computer Society.