Managing Variability Using Heterogeneous Feature Variation Patterns

Imed Hammouda, Juha Hautamäki, Mika Pussinen,
and Kai Koskimies

Institute of Software Systems, Tampere University of Technology,
P.O.BOX 553, FIN-33101 Tampere, Finland
{firstname.lastname, juha.o.hautamaki}@tut.fi

Abstract. Feature-driven variability is viewed as an instance of multi-dimensional separation of concerns. We argue that feature variation concerns can be presented as pattern-like entities - called feature variation patterns - cross-cutting heterogeneous artifacts. We show that a feature variation pattern, covering a wide range of artifact types from a feature model to implementation, can be used to manage feature-driven variability in a software development process. A prototype tool environment has been developed to demonstrate the idea, supporting the specification and use of heterogeneous feature variation patterns. We illustrate the idea with a small example taken from J2EE, and further study the practical applicability of the approach in an industrial product-line.

1 Introduction

The software engineering community is becoming increasingly aware of the nature of software systems as multi-dimensionally structured collections of artifacts: no single structuring principle can cover all the possible concerns of the stakeholders of a software system. This observation has far-reaching implications on how we construct, understand and manage software systems. Multi-dimensional approaches to software engineering have been the target of active research for a long time [1, 2, 3].

In the context of software product-lines [4, 5], one of the central concerns is variability management. The aim of variability management is to change, customize or configure a software system for use in a particular context [6]. In feature-driven variability management, variations of software products are expressed in terms of feature models (e.g. [7]). Selections of certain variants in a feature model are reflected in the design and implementation of the resulting product. Thus, a feature and its variation points constitute a slice of the entire system, cross-cutting various system artifacts ranging from feature models to implementation. Although variability management has been recognized as one of the key issues of software product-lines, its tool support lacks systematic approaches: most tools used in the industry are specific to a particular domain or product-line platform. Typically, automated support for variability management

M. Cerioli (Ed.): FASE 2005, LNCS 3442, pp. 145–159, 2005.

is based on product specifications given in, say, XML, used to generate the actual product by a proprietary tool.

A particular challenge for more systematic tool support for variability management is the fact that variability concerns span different kinds of artifacts and different phases of the development process. Even if we forget informal documents, the artifacts involved in variability concerns may include formal requirement specifications, design models, Java source files, XML files, scripts, make files, etc. The languages these artifacts are expressed in vary from graphical notations like UML [8] to various textual languages. Within UML, so-called profiles can be further used to create specialized modeling languages as extensions of UML for various purposes. Thus, we need a tool concept for variability concerns which is easily adapted to any reasonable artifact format.

In our previous work [9, 10, 11, 12] we have studied how a generic pattern concept can be used as a basis of tool support for various cross-cutting concerns like framework's specialization interfaces, maintenance concerns, and comprehension concerns within artifacts of a particular kind (e.g. UML design models or Java source code files). Here the term pattern [1] refers to a specification of a collection of related software entities capturing a concern in a software system; a pattern consists of roles which are bound to the concrete entities.

In this paper we generalize the pattern concept to allow multiple artifact types within the same pattern, thus satisfying the needs of feature variation patterns. We argue that the pattern concept is particularly amenable to present such heterogeneous patterns, since the basic pattern mechanisms are independent of the representation format of the artifacts, as long as there is a way to bind certain elements appearing in the artifacts to the roles of the pattern. This is in contrast to traditional aspects [13] which are presented using language-dependent mechanisms. The main contributions of this paper are the following:

- An approach to provide tool support for representing concerns within heterogeneous artifact types covering different phases of the development process
- The concept of a feature variation pattern as a model for tool-supported feature-driven variability management
- A prototype tool environment allowing the specification and use of feature variation patterns, together with early experiments

We proceed as follows. In the next section we briefly sketch the main idea. In Section 3, we explain the basic structuring device our approach is based on, the pattern concept. In Section 4, we give an overview of our current prototype environment supporting heterogeneous patterns. In Section 5, we illustrate our approach with an example concerning feature variation management in the J2EE environment. In Section 6, we present a small case study where we have applied the idea of feature variation patterns to a product-line provided by our industrial partner. Related areas in software engineering are discussed in Section 7. Finally, we summarize our work in Section 8.

[1] Our pattern concept has little to do with, say, design patterns: a pattern is a low-level mechanism that can be used to represent a design pattern or some other concern.

2 Basic Idea

We propose that feature variability can be managed using an artifact-neutral structuring device, a feature variation pattern. We will explain the pattern concept in more detail in the next section. Basically, a pattern consists of roles which are bound to actual system elements located in various artifacts; the pattern defines the required relationships between the elements bound to its roles. A feature variation pattern collects together elements relevant for realizing the anticipated variability of a feature across multiple artifacts. Ranging from requirements descriptions to actual implementation, these artifacts may be created in different phases of the software development process, and manipulated by different tools.

A single tool is used to manage the patterns, communicating with artifact-specific tools through their APIs. The existence of a feature variation pattern makes it possible for the tool to guide the product developer in exploiting the variability provided by a product-line platform, to assist in the generation of product-specific parts of the system, to make sure that the product has been developed according to the assumptions of the platform, to trace design-level variability support back to requirements, and to extract a system slice representing a single feature variation concern. This paper focuses on the two first issues. The general idea of the tool concept is illustrated in Figure 1.

Our approach is conservative in the sense that we do not make assumptions about languages or design methods: the only thing we assume is that the tools used to process the relevant system artifacts offer an API which allows the pattern tool to access the elements of those artifacts and to catch certain events (e.g. when an artifact has been modified). This assumption holds for many modern tools; in this work we have used Rational Rose for UML models and Eclipse for Java and XML. The artifacts can be freely edited through their dedicated tools: if an artifact is modified, the worst that can happen is that some bindings in an existing pattern become invalid or certain constraints defined by the pattern are violated. In this case, the pattern tool warns the developer about the inconsistencies and proposes corrective actions. It is then up to the developer to either correct the situation or ignore the warning. A prototype tool environment is presented in more detail in Section 4.

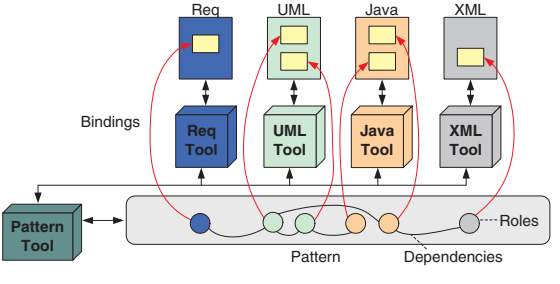

Fig. 1. The pattern-based approach to managing feature variability

3 Patterns

We view a feature variation pattern as an organized collection of software elements capturing any concern in a software system related to the variation of a particular feature. The target system could be a software product-line, or a single product with anticipated needs for feature variation.

Figure 2 depicts a conceptual model in UML for (feature variation) patterns. A pattern is a collection of hierarchically organized *roles*. A pattern is instantiated in a particular context by binding the roles to certain elements of concrete artifacts. Each role can be associated with a set of *constraints* expressing conditions that must be satisfied by the element bound to a role.

Artifacts contain models which can be expressed in different notations following well-defined metamodels. Here we regard any formal presentation describing some system properties as a model, including, say, UML models and source code. A metamodel is assumed to define a containment relationship between the model elements. In any binding, the containment relationships of the bound elements must respect the hierarchy of the roles.

The metamodels of the notations used in a model define properties for the model elements that can be checked by constraints. Constraints may refer to the elements bound to other roles, implying *dependencies* between the roles. For example, in a pattern which covers UML models, a constraint of an association role may require that the association bound to this role must appear between the classes bound to certain class roles, thus implying a dependency from the association role to the two class roles.

A role is associated with a *type*, which determines the kind of model elements that can be bound to the role. A role type typically corresponds to a metaclass in the metamodel of a given notation. As indicated in the lower part of Figure 2, a pattern can be associated with multiple sets of role types (for example UML, Java, etc). Each set groups together related role types. For example, there is a set

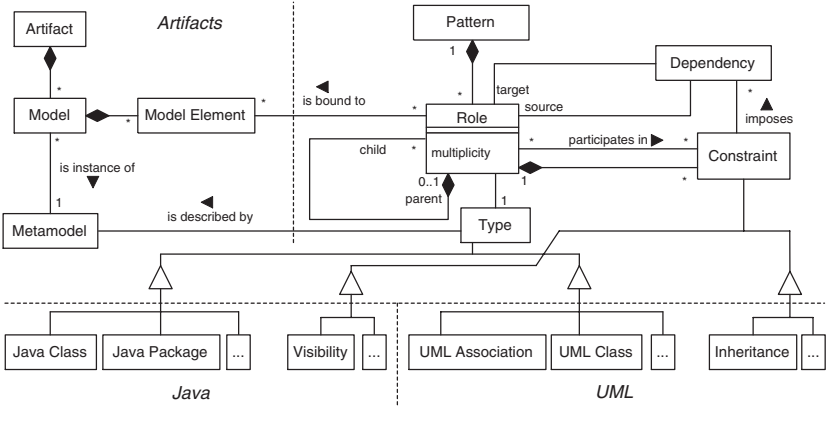

Fig. 2. Conceptual model for feature variation patterns

of role types for representing UML model elements. In this paper, we use patterns with role types covering a subset of UML (for representing feature models and design models), Java (for representing the actual implementation), and general text (for representing deployment descriptors). Each set of role types can be associated with specialized constraints applicable only for the roles in that set. For example, a Visibility constraint checks the visibility option of classes and their members in Java.

A *multiplicity* is defined for each role. The multiplicity of a role gives the lower and upper limits for the number of elements playing the role in an instantiation of the pattern. For example, if a class role has multiplicity [0..1], the class is optional in the pattern, because the lower limit is 0.

4 A Prototype Environment - MADE

Our experimental environment supporting feature variability management is the result of the integration of several existing tools. Eclipse [14] is used as a platform for JavaFrames [9] that implements the previously described generic pattern concept for Java role types. Eclipse acts also as a Java IDE in our work. The pattern engine of JavaFrames has been lately exploited in the MADE tool for creating a pattern-driven UML modeling environment [12]. This has been done by adding UML specific role types and integrating the resulting UML pattern tool with Rational Rose [15]. We have further extended the pattern tool with simple text file role types, allowing text files or their tagged elements to be bound to the roles as well. Thus, we can bind the roles to, say, XML files or items.

The MADE tool transforms a partially bound pattern into a task list. This is done by generating a task for each unbound role that can be bound in the current situation, taking into account the dependencies and multiplicities of roles. By performing a task, the designer effectively binds a role to an element. In order to use patterns as a generative mechanism (as in this paper), a *default element* can be defined for every role. If a role with a default element specification is to be bound during the pattern instantiation process, the binding can be carried out by first generating the default element according to the specification, and then binding the role to this element. The pattern engine updates the task list after a task has been performed, usually creating new tasks. When updating the task list, the pattern engine also checks that the constraints of the roles are satisfied, and generates corrective tasks if this is not the case.

A main principle in the design of our environment has been avoiding any kind of compulsive working mode of the designer. The existence of patterns does not prevent normal editing of, say, UML models or Java code; the purpose of the patterns is to offer additional support rather than a strait-jacket. If some constraint of a pattern bound to an element (or the binding itself) is broken as a result of an editing action, the tool generates immediately a new task to repair the broken constraint or rebind the role. In many cases the tool can even offer an automated repairing option.

5 Illustrative Example: Managing Persistence in J2EE

We illustrate the idea of feature variation patterns with a simple J2EE application. A typical J2EE application makes use of a persistent data storage which can be realized by different database products. We assume that the developers of the application want to make it easy to select the most suitable database solution for different customers. Thus, the possible variations of the database solution are specified in the feature model, and the design is given in such a way that all the desired variations can be easily achieved.

Assuming bean-managed persistence, the bean should be able to decide for optimization reasons which data source implementation to use. After the right implementation has been established, the bean can rely on a standard interface (DAO, Data Access Object) implemented by all the different data sources. There are two common solutions to select a data source implementation. The first is to hardcode the name of the implementation class in a specific method in the bean. When the data source changes, the implementation of that method should be changed so that it would return the proper implementation class. Another solution is to store the name of the implementation class in the deployment descriptor of the bean, as an environment variable. The bean decides at run-time which data source to use by looking up the value of this environment variable.

In both techniques, either the Java code or the deployment descriptor should change according to the data source selected. However, even if the developer decides to hardcode the implementation class in the bean code, storing the information of the used data source in the deployment descriptor might serve other purposes such as application documentation. In addition, the design model of the application should change in the sense that a class corresponding to the selected data source is added to the model filling the variation point.

5.1 Representing Database Selection as a Feature Variation Pattern

The above situation can be described using a feature variation pattern. The pattern has roles representing concrete elements at four abstraction levels: feature model, design model, Java source code, and deployment descriptor (XML). The pattern is given using a dedicated editor in MADE; however, we illustrate the pattern in Figure 3 with a dependency graph. The four different artifact types are represented by circular shapes. Roles are denoted by rounded rectangles, with role type marks (<<role type>>). Prebound roles are shaded. Role dependencies are drawn as broken arrows, while containment relationships are presented with diamond edges.

In the feature model part, there is a role named 'ConcreteDatasource'. This role represents the data source variant to be used. In the design model part, the UML class role named 'BeanDAOImplementation' stands for the model element indicating the proper DAO implementation. This role should be bound to a UML class in the application design model. Role 'Implementation' is used to reflect the data source decision at the code level. This role should be bound to a Java

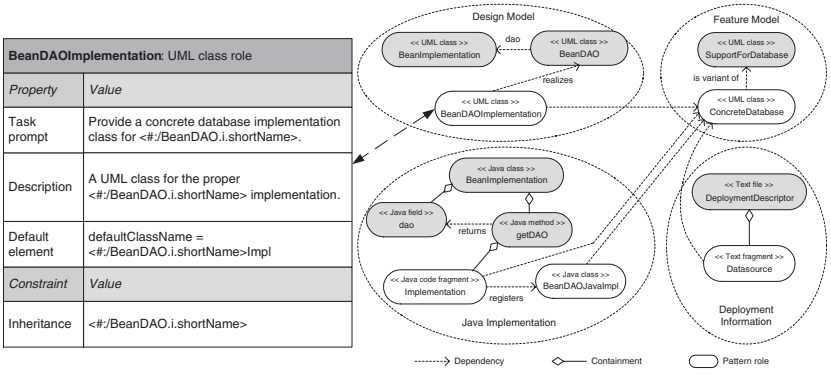

Fig. 3. Pattern role diagram

code fragment that specifies the proper DAO Java implementation. Furthermore, there is a role for storing the data source decision in the deployment descriptor of the bean. This role is named 'Datasource' and should be bound to a text fragment providing the right descriptor XML tags. All these roles depend on the database variant and are bound during the specialization process.

In addition to these roles, the pattern defines prebound roles specifying the context of the above roles. The 'DeploymentDescriptor' role, for instance, is bound to the deployment descriptor file where the generated XML text fragment should be inserted. Similarly, the Java class role 'BeanImplementation' represents the bean implementation class where the concrete DAO implementation should be registered.

There is a dependency from the four roles 'Datasource', 'Implementation', 'BeanDAOJavaImpl', and 'BeanDAOImplementation' to 'ConcreteDatasource' since the concrete elements bound to these four roles depend on the chosen data source implementation. The pattern tool generates the tasks following the partial order defined by the dependency relationships of the roles.

The actual pattern specification defines a set of tool-related properties for each role such as the task prompt, an informal description of the task (shown to the user together with the task prompt), and the possible default element generated prior to binding. The table in Figure 3 illustrates how these properties of the roles are specified in MADE. The table presents the properties of role 'BeanDAOImplementation'. Note that the specifications refer to the names of the elements bound to other roles using the $< \# : ... >$ notation, an expression of a simple scripting language used in our tool.

Due to such references, the values of these textual properties of the role are adapted to the current binding situation; for example, the task prompt and task description always use the application-specific names. In this case the default element is a template for the UML class of the proper DAO implementation. The template gives the name of the DAO implementation class and refers to the concrete name of the DAO class. In addition to role properties, the table in Figure 3 includes an example inheritance constraint which is used to check the

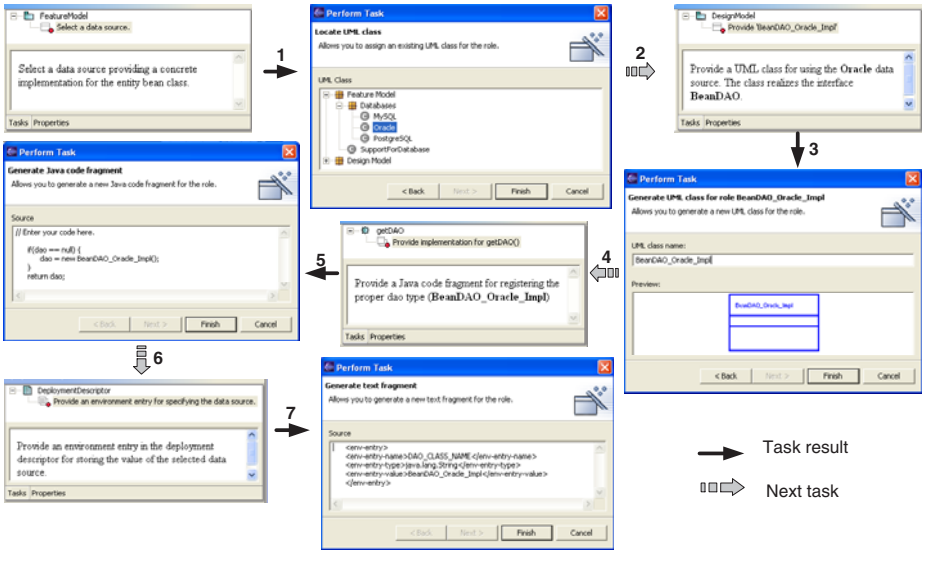

Fig. 4. Pattern binding steps

generalization/specialization relationship between the UML class bound to role 'BeanDAOImplementation' and the UML class bound to role 'BeanDAO'.

5.2 Using the Pattern

Figure 4 shows a scenario for applying the pattern. The MADE tool transforms an unbound role into a task, shown as a textual prompt. The execution of the task results in binding the role. The figure includes four tasks and their outcome.

The scenario starts from the left upper corner. First, a task prompt asks the user to select the data source to be used. The user is shown the list of available data sources: MySQL, Oracle, and PostgreSQL (1). The user decides to use the Oracle database. Next, a new task for providing a UML class named 'BeanDAO_Oracle_Impl' is shown (2). The UML class stands for the implementation class of the DAO interface. Note that the environment adapts the task description to the context of the user: the selected database name 'Oracle' is used in the default name of the UML class (3). The next task is to register the DAO Java implementation class to be used by the bean (4). A Java code fragment is then generated (5). The code creates a new instance of class 'BeanDAO_Oracle_Impl' and assigns it to the bean field holding the DAO object. Finally, the last task is to store the name of the DAO implementation class in the deployment descriptor of the bean (6). For this purpose, a new environment entry 'DAO_CLASS_NAME' is generated. The value of the entry is 'BeanDAO_Oracle_Impl' (7).

Figure 5 shows an overall view of our environment after the tasks described above have been carried out. In the upper half of the screen, Rose displays the feature model (on the left) and the design model (on the right) as UML class diagrams. In the lower half of the screen, the Eclipse Java environment displays

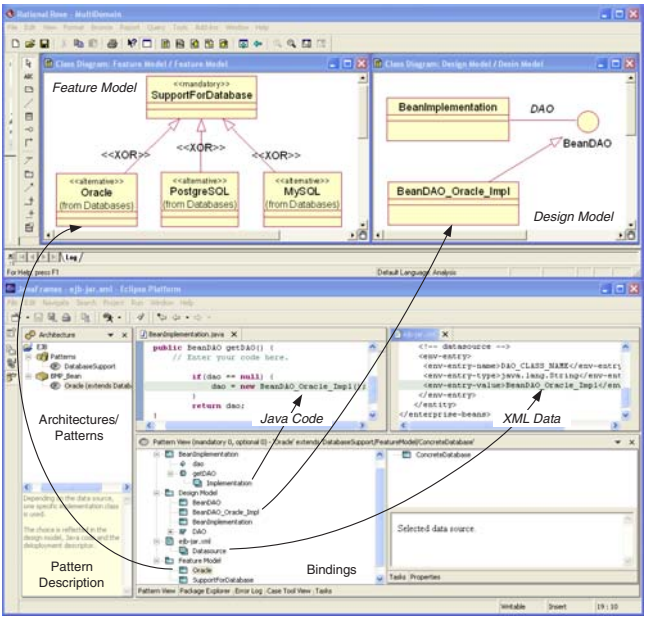

Fig. 5. A prototype environment

two textual editors (above) and the integrated pattern tool (below). The latter is further divided into three panes: the pattern view showing the roles in a containment tree (left), the task view (upper) and the instruction view (lower). The textual editors are for Java code and for XML.

When a pattern is selected in the pattern view, the task view displays the tasks generated by that pattern. In Figure 5, however, no doable tasks are shown since the pattern is fully bound. Instead, the view reveals a description of a task (data source selection) that has already been carried out. The outcome of every performed task can be retrieved in this way at any time. The bindings, visualized with the arrows in Figure 5, show how a pattern acts as a connecting artifact between different model levels.

6 Case Study - Nokia GUI Platform

6.1 Target System

Nokia produces a family of NMS (Network Management System) and EM (Element Manager) applications, which are software systems used to manage networks and network elements. For this purpose, the company has developed a Java GUI (Graphical User Interface) platform to support the implementation of the GUI parts for the variants of this product family. The platform is used as an object-oriented Java framework, in which the application is created by deriving new subclasses and by using the components and services of the framework.

The platform provides a number of services useful for GUI applications. There are services for system logging, online help, user authentication, product internationalization, clipboard usage, CORBA-based communication, and licensing. Depending on the environment used, each service may have different implementations. Applications, built on top of the GUI platform, get a reference to the proper service implementation from a registry file. It is essential that product developers register the right service implementation. Due to the confidential nature of the platform, detailed descriptions about its architecture are omitted.

6.2 Experiment

The goal of the case study was threefold: identifying variability in the GUI platform, expressing the identified variation points in terms of feature models, and using our pattern concept and prototype tool to achieve an environment where variability is managed across multiple artifacts.

Our first step was to analyze the platform documentation and interview several of the platform users. As a result, we have identified a number of variability issues regarding how platform services are being used. The next step was to construct feature models realizing the variability issues we have identified. Figure 6 depicts, in lighter color, a feature model representing the main services provided by the GUI platform. All services are optional. It is up to the product developer to decide which services to use. The 'I18n' service stands for the internationalization service.

As a third step, we have developed a system of feature variation patterns for expressing which platform services are used and which service implementations are considered. Because a service is regarded as a separate concern in the platform, a pattern (or a set of related patterns) is used to represent that service. The patterns cover four abstraction levels: the feature model representing the services, the design model of the product, the Java implementation of the application, and the product service registry files.

When instantiating the patterns, developers decide which services the application should incorporate and which service implementation to use. If a service is selected, MADE uses the corresponding pattern to generate tasks for registering the service in the application service registry. Furthermore, the pattern ensures that the right service implementation is added to the design model of the application, that the Java implementation exists, and that the right (and only one per service) service implementation is registered to the property files.

Each of the services shown in Figure 6 in lighter color can be further represented by its own feature models. The figure, for example, depicts, in darker color, a feature model specifying how application GUI components can use the on-line help service. First, the variant 'User_Event' indicates that the call resulted from a user request whereas 'System_Event' indicates that the call originated from the application. GUI components can support either event types but not both at the same time. Second, developers must specify what type of help service is requested. There are five variants for such service type. 'Contents', for example indicates that the target of the request is a table of contents whereas 'Search'

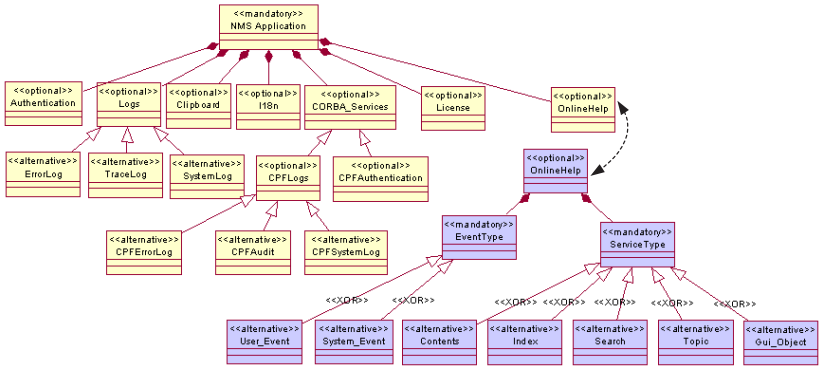

Fig. 6. Feature models for platform services and online help service

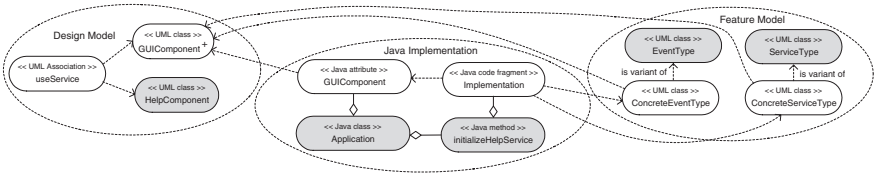

Fig. 7. Role diagram for the OnlineHelp pattern

indicates that the target of the request is a search page. Similarly, the choice of the service type is exclusive.

A feature variation pattern is used for representing the feature model for online help. The pattern is used to generate tasks for selecting the GUI components which incorporate a help service and the help event and service types associated with those components. In addition, it ensures that only one variant is selected and that it is correctly represented in the Java implementation. Furthermore, the pattern is used to associate, in the design model, the help service with the selected GUI components.

The role structure of the pattern is shown in Figure 7. The two roles named 'ConcreteEventType' and 'ConcreteServiceType' represent the event and service type variants to be used. In the design model, role 'GUIComponent' should be bound to a UML class representing the GUI components associated with the help service. The '+' symbol in front of the role name stands for the multiplicity value meaning, in this case, that there can be any number of GUI components (UML classes) bound to the role. At the implementation level, the pattern uses the variants in the feature model to generate a Java code fragment for registering the proper help service to the selected GUI components. This is illustrated by role 'Implementation'. The code fragment is inserted in an initialization method of the application.

6.3 Experiences

As explained earlier the goals of the case study have been to study the expressive power of our formalism and the applicability of our approach in an industrial setting. Our first challenge has been to dig up the feature models from the platform documentation, where the documents were not structured according to our methodology. Another problem, during this phase of the experiment, was the fact that the platform is used by different groups in the company having different interests towards the product line. Thus, we had to interview each of these parties. Furthermore, the platform comes with different versions making it harder to identify the variation points.

The variability aspects of the platform were mostly specified in Word documents. An attractive option would have been to link the relevant parts of these documents to the feature variation patterns, rather than (or in addition to) feature models. However, this would require different structuring of the Word documents and new role types covering elements in these documents. In the case study, we have specified all platform services as optional features, even though services such as internationalization or online help are in practice required. At this stage, we could conveniently present a number of variation points in the platform with a set of heterogeneous patterns. However, we still need to construct more patterns in order to cover variability in other platform components.

7 Related Approaches

Feature Variability Management
One of the key issues in software product lines is variability management. A product line architecture makes it easier to manage the product family as it promotes the variation between different products, i.e., the use of variants and variation points [16]. The software community has taken different approaches to variability management. Our methodology is based on feature models [7]. Other methodologies include architecture description languages (ADLs) [17] and different XML-based program specification [18].

Framed aspects [19] are another approach for representing features in software product lines. The purpose of the method is to support evolution in product lines rather than the development of products. First, aspects are used to encapsulate tangled features. Then, frames are used to provide parameterization and reconfiguration support for the features. Compared to our approach, framed aspects are applied at the implementation level only.

Model-Driven Engineering
MDD (Model-Driven Development) [20] promotes an approach where models of the same system are usually derived from each other leading to better alignment between the models. MDA (Model-Driven Architecture) [21] is a recent initiative by OMG for supporting MDD principles. MDA defines three views of a system: a Computation Independent Model (CIM), which is a representation of a system from a business viewpoint, a Platform Independent Model (PIM), which is a rep-

resentation of a system ignoring platform specific details, and Platform Specific Model (PSM), which is a model of a system that covers both platform independent information and details about a specific platform. Compared to MDA, our feature models correspond to CIM whereas other model kinds can be viewed as PSM. In this paper we do not discuss support for PIM. An MDA-oriented approach, using our pattern concept, is presented in [22].

Batory et al. take an approach to feature oriented programming where models are treated as a series of layered refinements [23]. Features are composed together in a step-wise refinement fashion to form complex models. Models can be programs or other non-code representations. To support their concepts, the authors have developed a number of tools for feature composition, called AHEAD toolset. The toolset provides similar functions to those of MADE. MADE environment solves two problems not otherwise addressed in [23]: tracing features across different artifacts and checking the validity of models.

Separation of Concerns Across Artifacts

The idea of representing concerns within and across different artifacts has been addressed in the work on multi-dimensional separation of concerns [2]. The authors present a model for encapsulating concerns using so-called hyperslices. These are entities independent of any artifact formalism. Our heterogeneous pattern concept can be considered as a concrete realization of the hyperslice concept. Other concepts such as subjects [24], aspects [1], and viewpoints [25] also resemble our patterns. Subjects are class hierarchies representing a particular view point of a domain model. Thus subjects deal only with object-oriented artifacts. Aspects, on the other hand, have been mostly used to represent concerns at the programming level. Viewpoints, in turn, are used to represent developers' views at the requirements level. Different viewpoints can be described using different notations. Compared to these concepts, our pattern approach is not bound to a specific artifact type but can be used to capture concerns cross-cutting different phases of the development process.

Tool Support for Traceability

The ability to track relationships between artifacts has been a central aim in requirements engineering [26]. Rational RequisitePro [27] is a market-leading requirement management tool. In RequisitePro, use cases are written as Word documents which are stored into a relational database. Use case diagrams can be associated with use case documents; sequence diagrams implementing the use cases can be linked to the use case diagrams, and class diagrams can be linked to the sequence diagrams. A related approach has been proposed to achieve traceability in software product families [28], linking requirements, architecture, components, and source code. In both works, traceability is based mainly on explicitly created links. In our approach there are no explicit links between the artifacts themselves, but instead we specify a particular concern as a pattern and bind the roles of the pattern to certain elements of the artifacts.

8 Concluding Remarks

We have developed a prototype environment supporting the representation of concerns cross-cutting not only components but also various artifacts produced in different phases of a software development process. Our first experiences with the environment are encouraging: we could conveniently present variation points in a product-line with a heterogeneous pattern covering multiple artifact types. Using existing tool technology for pattern-driven software development, we could achieve an environment where the pattern guides variation management from feature model to actual implementation.

However, feature variation management is just one example of the potential benefits of our environment. The main point is that the pattern stores the information of the existence of a concern among the artifacts. This information can be exploited in many ways. For example, it is often useful simply to generate a single view where all the fragments related to a particular concern can be browsed, a kind of concern visualizer. This kind of support is readily available in our tool. Heterogeneous patterns can be used for tracing as well: the designer can follow the dependencies between the roles of a pattern and find out why, for example, a particular class has been introduced in the design model.

In order to support new representation formats of artifacts, we are working at transforming the tool into a framework for constructing new role types. Other future directions include enhancing the alignment and traceability between the various artifact types, and proceeding with the case studies provided by our industrial partners.

Acknowledgments

This work is supported financially by the Academy of Finland (project 51528) and by the National Technology Agency of Finland (projects 40183/03 and 40226/04), and is partly (first author) funded by Nokia Foundation.

References

1. Elrad, T., Filman, R.E., Bader, A., Editors: Communications of the ACM. Special issue on Aspect-Oriented Programming, 44:10 (2001)
2. Tarr, P., Ossher, H., Harrison, W., and Sutton, Jr., S. M: N degrees of separation: Multi-dimensional separation of concerns. In: Proc. ICSE 1999, Los Angeles, CA, USA, ACM Press (1999) 107–119
3. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM **15** (1972) 1053–1058
4. Bosch, J.: Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach. Addison.Wesley (2000)
5. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2002)
6. Van Gurp, J., Bosch, J., Svahnberg, M.: On the notion of variability in software product lines. In: Proc. WICSA 2001, Amsterdam, The Netherlands (2001) 45–55

7. Czarnecki, K., Eisenecker, U.: Generative Programming, Methods, Tools and Applications. Addison Wesley (2000)
8. OMG: UML WWW site. At URL `http://www.uml.org/` (2005)
9. Hakala M., Hautamäki J., Koskimies K., Paakki J., Viljamaa A., and Viljamaa J: Generating application development environments for Java frameworks. In: Proc. GCSE 2001, Erfurt, Germany, Springer, LNCS 2186 (2001) 163–176
10. Hammouda, I., Harsu, M.: Documenting maintenance tasks using maintenance patterns. In: Proc. CSMR 2004. (2004) 37–47
11. Hammouda, I., Guldogan, O., Koskimies, K., Systä, T.: Tool-supported customization of uml class diagrams for learning complex system models. In: Proc. IWPC 2004, Bari, Italy (2004) 24–33
12. Hammouda, I., Koskinen, J., Pussinen, M., Katara, M., Mikkonen, T.: Adaptable concern-based framework specialization in UML. In: Proc. ASE 2004, Linz, Austria (2004) 78–87
13. Eclipse: AspectJ WWW site. At URL `http://eclipse.org/aspectj/` (2005)
14. Eclipse: Eclipse WWW site. At URL `http://www.eclipse.org` (2005)
15. IBM Rational: Rational software. At URL `http://www-306.ibm.com/software/rational/` (2005)
16. Jacobson, I., Griss, M., Jonsson, P.: Software Reuse - Architecture, Process and Organization for Business Success. Addison-Wesley (1997)
17. Van der Hoek, A.: Capturing product line architectures. In: Proc. ISAW-4, Limerick, Ireland (2000) 95–99
18. Cleaveland, J.: Program Generators with XML and Java. Prentice-Hall (2001)
19. Loughran, N., Rashid, A., Zhang, W., Jarzabek, S.: Supporting product line evolution with framed aspects. ACP4IS Workshop AOSD 04 (2004)
20. Mellor, S., Clark, A., Futagami, T.: Model-driven development. IEEE Software **20** (2003) 14–18
21. OMG: MDA guide version 1.0.1. At URL `http://www.omg.org/docs/omg/03-06-01.pdf` (2003)
22. Siikarla, M., Koskimies, K., Systä, T.: Open MDA using transformational patterns. In: Proc. MDAFA 2004, Linköping, Sweden (2004) 92–106
23. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling stepwise refinement. In: Proc. ICSE 2003, Portland, USA (2003) 187–197
24. Clarke, S., Harrison, W., Ossher, H., and Tarr, P: Subject-oriented design: towards improved alignment of requirements, design, and code. ACM SIGPLAN Notices **34** (1999) 325–339
25. Nuseibeh, B., Kramer, J., Finkelstein, A.: A framework for expressing the relationships between multiple views in requirements specifications. Transactions on Software Engineering **20** (1994) 760–773
26. Leffingwell, D., Widrig, D.: Managing Software Requirements. Addison.Wesley (2000)
27. IBM Rational: Rational RequisitePro. At URL `http://www3.software.ibm.com/ibmdl/pub/software/rational/web/datasheets/version6/reqpro.pdf` (2005)
28. Lago, P., Niemel, E., Van Vliet, H.: Tool support for traceable product evolution. In: Proc. CSMR 2004, Tampere, Finland (2004) 261–269