# Supporting Model Reusability with Pattern-based Composition Units

Andrey Nechypurenko
*Siemens Corporate Technology*
*Munich, Germany*
*andrey.nechypurenko@siemens.com*

Douglas C. Schmidt
*Vanderbilt University,*
*Nashville, TN, USA*
*d.schmidt@vanderbilt.edu*

## Abstract

*The growing complexity and criticality of distributed systems motivates software developers to raise the level of abstraction used to develop these systems. A promising approach for improving the quality and productivity of software development is to (1) assemble applications from higher-level building blocks that represent solution templates for certain application domains and (2) apply model-driven development techniques and tools to manipulate the building blocks and automate key tasks related to system specification, implementation, configuration, and deployment, rather than (re)writing the applications manually using third-generation programming languages. To simplify the manipulation of component building blocks, however, requires a well-formed set of rules and relationships. This paper contributes to the study of these topics by describing pattern feature inheritance relationships, showing how pattern feature inheritance can improve the reusability of models, and illustrating our approach with a concrete example adding the remoting aspect to a GUI application.*

## Keywords

Model-Driven Development (MDD), patterns, inheritance.

## 1. Introduction

**Emerging trends**. The growth in the size and complexity of large-scale distributed systems is exceeding the ability of IT professionals and organizations to develop software for these systems with acceptable and affordable time and effort. To address this problem requires new technologies that enable developers to improve the productivity and quality of the software development process. A promising approach involves the combination of (1) *component middleware* [21], which provide mechanisms to configure and control key distributed computing aspects (such as connecting event sources to event sinks and managing transactional behavior) separately from the functional aspects of applications, with (2) *model driven development (MDD)* [1][22], which are generative technologies that help reduce complexity by raising the level of abstraction at which software is developed, validated, and disseminated.

The technical foundations of component middleware consist of various patterns and frameworks that have been covered extensively in earlier publications [4][5][10][14][17]. The technical foundations of MDD are less well codified, but the emerging consensus [11][22] is that the MDD paradigm involves (1) *metamodeling*, which define type systems that precisely express key characteristics and constraints associated with particular application domains, such as e-commerce, telecommunications, and automotive control, (2) *domain-specific languages*, which provide programming notations that formalize the process of specifying business logic and quality of service (QoS)-related requirements, and (3) *model transformations and code generators*, which help to automate and assure the consistency of software implementations using analysis information associated with functional and QoS specifications captured by models. Although there are various approaches [1][20] to realizing the MDD paradigm, MDD tools and techniques share a common goal of reducing complexity by raising the level of abstraction used to specify, implement, configure, and deploy software systems.

**Unresolved problems**. Despite improvements in third-generation programming languages (such as Java or C++) and run-time platforms (such as component and grid middleware), the levels of abstraction at which application logic is typically integrated with the set of rules and behavior dictated by conventional design and programming techniques remains low relative to the (1) concepts and concerns within the application domains themselves and (2) advanced technologies available in the solution space, as described below:

- **Gap between domain and implementation abstraction levels**. A large gap exists in the levels of abstraction between (1) third-generation programming languages used by software engineers versus (2) the domain-specific terminology used by systems engineers to describe applications that are being built. The conventional solution is to apply a design process (such as object-oriented design or structured design) to map from the higher-level domain-specific abstractions to the much lower-level abstraction provided by mainstream third-generation programming languages. This mapping has historically been performed manually by conventional software development methodologies, such as RUP [25], which introduce various problems, ranging from simple implementation errors to missing customer requirements [22].

- **Gap between state-of-the-art and state-of-the-practice**. Another gap exists between the levels of abstraction and composition that represent (1) the state-of-the-art in software engineering R&D versus (2) the state-of-the-practice applied by most developers. In particular, third-generation languages do not intuitively reflect the concepts used by cutting-edge software researchers and developers [9], who increasingly express their system architectures and designs using languages and tools that directly support higher-level concerns, such as persistence, remoting, and synchronization.

Both these gaps can be narrowed by introducing intermediate abstraction layers that reduce the distance between problem domain abstractions and available solution domain abstractions. As discussed in [22], this approach motivates the development of generative MDD technologies that create families of domain-specific languages (DSLs). These DSLs can then be applied to express domain-specific problems more effectively and intuitively than general-purpose third-generation programming languages, thereby enhancing software productivity and quality.

Despite the promising benefits of MDD, however, other unresolved problems remain due to the fact that models of distributed systems can themselves be large and complex as applications grow in size and scope. In particular, it is hard to change and maintain models using conventional Model-Driven Architecture (MDA) techniques [1][20]. which provide only a slightly higher level of abstraction and platform-independence than third-generation programming languages, such as C++ or Java. As a result, the gap between expressing problems in the domain space and representing them in the solution space remains too large.

**Solution approach ⇨ Compose software systems from higher-level building blocks that are solution templates for certain problems.** In previous work [2][3] we motivated the need for higher-level MDD abstractions that combine patterns, component middleware, and aspect-oriented software development (AOSD) techniques [13] to

- Resolve recurring distributed system development problems so they have fewer dependencies on platform-specific details, such as communication protocols, object models, and threading models, and
- Automate key system evolution tasks, such as implementing new customer requirements, refactoring certain parts of the system, and migrating to the newer versions (or versions from other vendors) of libraries and middleware used for development.

Our previous work, however, does not show how the pattern-based composition of different aspects and models could be implemented in component-based systems. This paper therefore explores another point in the solution space: *illustrating a new design and problem decomposition approach that applies patterns for modeling different aspects of distributed systems to simplify model transformations and code generators for component-based systems*. In particular, we investigate the relationships between patterns that can improve their substitutability and composability, thereby contributing to methodologies that can be applied to manipulate role-based solution templates as first class system composition units. We introduce the concept of *pattern feature inheritance relationships* and use a concrete example to illustrate the benefits of using the substitutability property of feature inheritance. It is our position that formalizing sets of composition and manipulation rules will enable greater automation of key modeling and code generation concerns that are hard to handle with conventional MDD technologies.

**Paper organization**. The remainder of this paper is organized as following: Section 2 describes how feature inheritance relationships between patterns help to support variability without degrading software symmetry [26][27]; Section 3 examines a concrete example that illustrates the applicability of concepts presented in Section 2 to solve the problems outlined in Section 1; Section 4 compares our approach with related work; and Section 5 presents concluding remarks and outlines future work.

## 2. Pattern Inheritance as a Key Mechanism to Encapsulate Variability and Improve Reusability

To manage software development effort and enhance software productivity and quality, the IT industry is continually trying to improve reusability and localize the impact of variability found in product families [8]. The paradigms developed over the past 3-4 decades range from functional decomposition to object-oriented decomposition and recently aspect-oriented decomposition [6][13]. Each paradigm prescribes a methodology for modularizing different dimensions of software systems. A theme that increasingly pervades all these software development paradigms is *patterns* [12][10][14], which are technology-independent, role-based descriptions of common ways to resolve key forces associated with recurring problems encountered when developing software.

Based on our experience developing and applying pattern-based [10][12][14] frameworks [4][5] and middleware platforms [16][17] for distributed systems over the past two decades, we believe that patterns are a valuable addition to the portfolio of higher-level system building blocks available to software developers. To enable patterns to become first-class citizens in MDD environments, it is necessary to define a set of composition rules and express relationships between patterns precisely. As discussed in [26], it is possible to substitute implementation artifacts that have inheritance relationships without affecting key properties of an

entire system. This type of transformation can be treated as a *symmetry* [29], which is a special type of model transformation that preserves the key properties of a model. Examples of model properties include *persistence*, which is the ability to read/write the state of an object to persistent storage and *remoting*, which is the ability to communicate with other system components over a network.

Coplien and Zhao [26] describe how object-oriented inheritance can also be treated as a symmetrical transformation because it preserves key behavioral aspects defined by base class. In turn, the concept of pattern feature inheritance introduced in this paper also preserves the key properties of the base pattern, so that substituting derived patterns provide variability without changing key system properties. This section describes how feature inheritance relationships between patterns help to support variability without degrading software symmetry. In particular, we treat transformation and inheritance as enabling mechanisms to simplify the substitution of certain system components without affecting other key system properties. These mechanisms also make it easier to handle the variabilities that are often encountered when developing MDD tools to support product-line architectures.

## 2.1 Handling Variability via Inheritance

Inheritance is a powerful mechanism for shielding certain parts of applications from side-effects caused by the need to customize certain functional aspects. To illustrate inheritance, consider the following classical Observer pattern [12] example shown in Figure 1. In this example, the **`Subject`** class is shielded from the variability introduced by different implementations of the **`Observer`** interface. The enabling mechanism in this case is *inheritance*, which in accordance to the Liskov Substitutability Principe (LSP) [7] allows **`Observable`** to work uniformly with all **`Observer`** subclasses, such as **`Notifier`** and **`Logger`**.

It would be nice to achieve the same level of substitutability with pattern-based building blocks. To achieve this functionality, therefore, we need to identify similar relationships between patterns. These relationships, in turn, should be used to facilitate the development of MDD tools that can automate pattern manipulation tasks and support the level of substitutability and transformation needed to address the challenges presented in Section 1.

## 2.2 Inheritance Relationships between Patterns

To explore the value of expressing inheritance relationships between patterns, we will examine the following set of patterns:

- **Observer** [12], which defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Reactor** [10], allows event-driven applications to demultiplex and dispatch service requests that are delivered to an application from one or more clients.
- **Interceptor** [10], which allows services to be added transparently to a framework and triggered automatically when certain events occur.

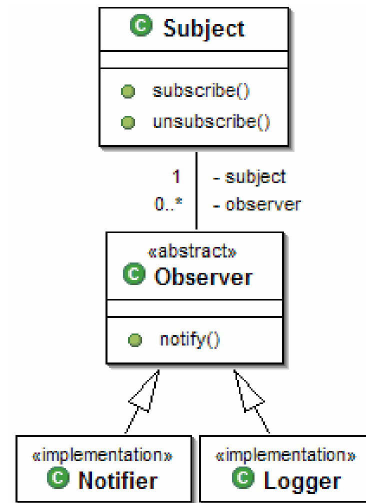


**Figure 1. Observer Pattern Structure**

There are common roles and responsibilities that cross-cut these patterns, e.g., there are certain events that can occur in a system, certain entities that need to be notified when such events occur, and certain ways these entities can express their desire to handle certain events by registering their interest. While this description characterizes the Observer pattern, it does not mean that Reactor and Interceptor are simply different variants of Observer since each pattern has different forces and goals. Yet there are also similarities that stem from the fact that these patterns share a higher-level relationship than just "different variants of Observer." We contend that this relationship can be represented by *feature inheritance*.

To explore feature inheritance relationships between patterns more concretely, consider again the Observer example presented in Section 2.1. The **`Notifier`** and **`Logger`** subclasses have different functionality and goals, i.e., notify users via a pop-up window and an output trace record, respectively, but they still conform to the "is a" relationship to the **`Observer`** base class. There are similar relationships for the Observer, Interceptor, and Reactor patterns, as shown in Figure 2.

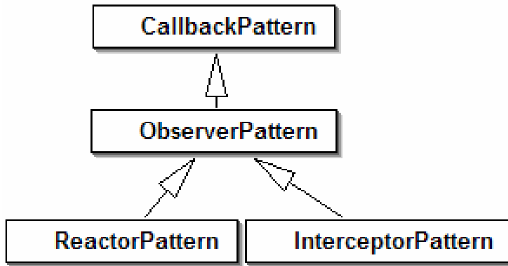


**Figure 2. Relationships between Patterns**

The pattern feature inheritance tree shown in Figure 2 defines the relationships between four patterns. At the root of the hierarchy is the **`Callback`** pattern [30], which defines the basic feature – control inversion – used by all the related patterns. In the **`Observer`** pattern, all registered **`Observers`** are called back by a registered **`Subject`**. Likewise, for the Reactor and Interceptor patterns the registered **`Event Handlers`** and **`Interceptors`** are called back, respectively, when the certain triggering conditions occur.

Despite the similarities between these four patterns, there are also some differences that bear mentioning because they motivate the **`Observer`** pattern as a basis for the set of related patterns and allow a cleaner connection between the patterns at Figure 2. In particular, a key difference between the Reactor and Interceptor patterns is the *event source*. The Reactor's event source is a demultiplexor, such as the **`select()`** or **`WaitForMultipleObjects()`** system calls, whereas the Interceptor's event source is an incoming control flow, such as callback method invocation by CORBA Portable Interceptors [28] that are triggered during the remote invocation request/response flow. There is, however, no such role as *event source* in classical Observer pattern description – instead, that role is merged with the *subject* role. We therefore suggest the Observer pattern be extended, as shown in Figure 3.

The new **`EventSource`** role shown in Figure 3 is responsible for monitoring possible condition changes and then initiating a notification propagation mechanism by triggering the **`Subject`** implementation, e.g., by invoking the **`triggerUpdates()`** method on the **`Subject`**. Introducing the **`EventSource`** role allows a cleaner separation of responsibilities for the Observer pattern. Moreover, compared with the previous approach shown in Figure 1, the **`Subject`** role is now only responsible for maintaining observers list and iterating over this list when notifications are propagated.

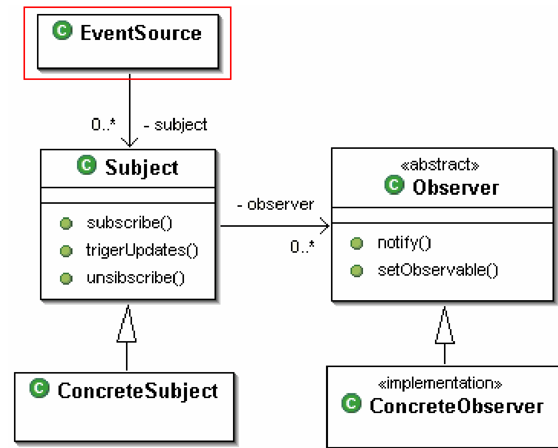


**Figure 3. Extending the Observer Pattern with the Event Source Role**

To illustrate different implementations of the **`EventSource`** role, the following cases could be considered:

- Different implementations of **`EventSource`**, e.g., example the family of different reactors, such as the **`ACE_Select_Reactor`**, **`ACE_WFMO_Reactor`**, and **`ACE_Dev_Poll_Reactor`** [5].
- A GUI event loop, which typically blocks on an OS demultiplexer, such as **`select()`** or **`WaitForMultipleObjects()`**, to detect incoming events (e.g., a mouse click) and then dispatch this event to the corresponding handlers (e.g., a button) , which in turn notifies observers about a change in state (e.g., button down).
- Hardware interrupt handlers can also be considered as event sources, which typically delegate event processing to observers in the OS kernel.

The Visitor pattern [12] could be also viewed as inheriting from Observer, where the event source is the traversing algorithm visiting various concrete nodes. For example, the Boost Graph Library (BGL) uses Observer pattern terminology (notify) for their generic visitor implementations of graph traversing algorithms [19].

We have identified other examples of inheritance relationships between patterns, as shown in Figure 4, which illustrates the set of patterns that solve similar problems using different methods.

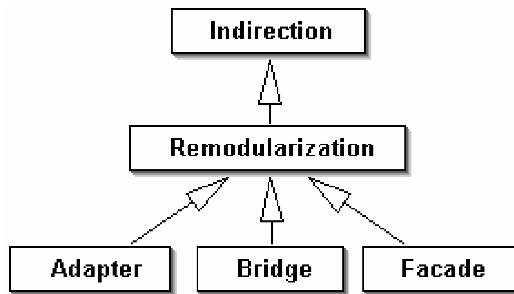


**Figure 4. Example of Inheritance Relationships Between Patterns**

Despite differences in structure and intent, the core mechanism used by these patterns in Figure 4 is the indirection between two collaborating parties, which is why the **`Indirection`** pattern forms the root of this feature inheritance tree. The second level in the tree shows the **`Remodularization`** pattern, which enables collaboration between two objects even if a mismatch occurs between a provided interface and an interface expected by a collaborator. In turn, there are different circumstances and types of remodularization required in each concrete case, which is why the three other patterns in Figure 3 are specializations of the **`Remodularization`** pattern.

## 2.3 Applying Feature Inheritance in Practice

Section 2.2 shows how feature inheritance relationships between concerns can be presented in the form of patterns or other role-based definitions. Using this concept, we can provide a powerful mechanism to encapsulate variability at a higher level of abstraction than is possible via third-generation programming languages. For example, we can encapsulate the impact of variability in the communication infrastructure (such as standard middleware or custom frameworks) on the rest of large-scale distributed systems.

The primary advantage of using feature inheritance in this manner is to systematically introduce changes to a system using roles defined by certain role-based solution descriptions. For example, if a developer wants to add a Visitor pattern implementation to the code, a wizard provided by an MDD tool could guide the user through the role mapping process to ensure that all roles defined by the Visitor pattern are mapped by the developer to the appropriate classes. The benefit of expressing feature inheritance relationships in this case is that after the mapping for the base pattern role is complete, subsequent substitutions of this pattern with concrete patterns can either be done automatically or semi-automatically (e.g., guided by wizards).

Figure 5 shows a high -level view of the complete modeling process described above.

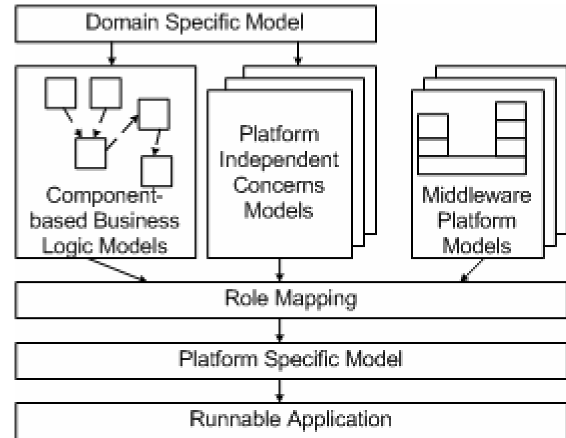


**Figure 5. Concern-based Modeling Process**

This figure shows how domain-specific models are used as an input for various modeling tools. Next, the set of predefined role-based solutions can be introduced by means of a role mapping step. Finally, after completing the role mapping process, platform-specific models can be generated, followed by a runnable application.

## 3. Remote Button Example

This section presents a concrete example that further illustrates how the approach presented in Section 2 could be applied in practice.

## 3.1 Scenario

Consider a standalone application that is based on the refactored Observer pattern shown in Figure 3. This application has a simple GUI in the form of dialog box with a single button. Pressing this button causes the invocation of a method that implements application-specific functionality. As shown in Figure 6, the button plays the **`Subject`** role in the Observer pattern and the application-specific class plays the **`Observer`** role (with the application-specific processing implemented in the **`Observer`**'s **`notify()`** method), and the GUI event processing loop plays the **`EventSource`** role.

Figure 6 also represents the mapping between roles defined by Observer pattern (i.e., **`Subject`**, **`Observer`**, and **`EventSource`**) and the application-specific classes (i.e., **`Button`** and the GUI event loop implementation). As a result of feature inheritance, the Observer pattern can be replaced with derived patterns without breaking the key functional properties of this example system, i.e., "business class should be notified whenever the button is pressed." This example illustrates how pattern feature inheritance supports transformation without breaking symmetry.

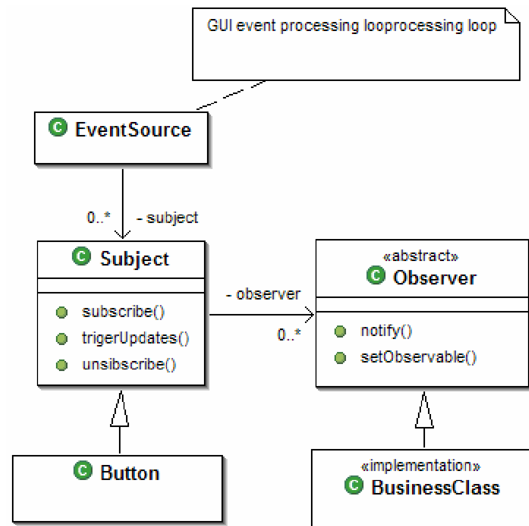


**Figure 6. GUI Example Structure**

## 3.2 Introducing the Remoting Aspect

The initial implementation of our GUI program shown in Section 3.1 was a standalone application. To work in a client/server environment, assume that the scenario's requirements change so that it is necessary to split this application in two parts that communicate across a network. The first part (i.e., the GUI client) will receive the push button event and then send this event over the network to the second part (i.e., the business server), which will then process this event the same way as in the initial scenario. After this substitution, sample GUI application will be split into two parts that communicate with each other across a network. We thus introduce the *remoting* aspect to the application, without changing key properties of the application, i.e., the **`BusinessClass`** will be notified when a button push event occurs.

We now analyze the impact of these changes on our initial application, in particular, on the server-side of the new client/server application. At one level, little has changed except for the event source, i.e., the source of the event notifications occurring in the system. In the standalone version, the event source was the GUI event loop that sent the mouse click notification to the standalone application. In the client/server configuration, conversely, the event source for the server-side will arrive from the network, i.e., the event source now is an OS demultiplexing, such as **`select()`** or **`WaitFor-MultipleObjects()`**.

Naturally, the **`Reactor`** pattern implementation is only part of the necessary interprocess communication (IPC) infrastructure. Introducing the remoting aspect for larger applications will therefore require more pattern implementations and associated aspects [17]. For the sake of clarity, however, this example assumes that the **`Reactor`** pattern implementation provides sufficient functionality to support our simple IPC infrastructure.

## 3.3 Substituting Observer with Reactor

Based on the discussion in Section 2.2, if the **`Reactor`** pattern inherits from the **`Observer`** pattern, we can substitute our Observer-based implementation with a Reactor-based implementation *without* affecting the business components, i.e., the **`Button`** and **`BusinessClass`** classes, which are written in terms of the **`Observer`** base class. The following list summarizes the steps made as a result of the substitution outlined above, focusing on the server-side modifications, which can be performed as follows:

1. Instead of running GUI event loop, the server needs to call the Reactor's **`run_event_loop()`** method, which will substitute the event source in the server application. Since this portion of the application is not part of the business logic and it will not require changes to application functionality, i.e., the implementation of Observer's **`notify()`** method by **`BusinessClass`** need not be changed.

2. The business logic implementation (i.e., the **`Observer`** role) contains registration logic (**`subscribe()`**) for events of interest. With the Reactor-based implementation the same step is required, i.e., event handlers should be registered with a reactor and need to pass an event mask that describes what types of events are of interest (e.g., the fact that there is data available in a socket). Once again, nothing should change in the application functionality.

3. The **`Observer`** (i.e., the event handler) will be notified by the reactor when there data is available in a socket registered with the reactor. After the reactor dispatches the handler, the handler can access the incoming data and perform the required processing steps.

Based on this analysis, it is clear that the processing steps for the original *application* functionality remain the same before and after adding the remoting aspect. In a larger example, it may also be desirable to devise a solution that affects as little of the *infrastructure* software as possible. The approach described above does not provide this level of transparency due to differences in the APIs used for various tasks, such as accessing the event attributes, which in the case of *GUI events* come from GUI toolkit supplied data structures associated with the event and in the case of *network events* come from a socket. There are ways to further enhance the solution to minimize code perturbation, including:

- Using a patterns-oriented software library that is designed for composition and thus using uniform meth-

ods for accessing notification information. For example, the ACE [4][5] and TAO [15, 16] middleware platforms could be applied to our example application to minimize infrastructure rework.

- Remodularize the base code using aspect-oriented techniques. For example, [9] proposes an approach that uses the notion of *collaboration interfaces* for remodularization of interfaces that were not designed to interact with each other initially.

We believe that the second approach is more flexible and will concentrate our future research work in this direction.

## 4. Related Work

This section reviews work related to our approach.

Generative programming (GP) [23] is a type of program transformation concerned with designing and implementing software modules that can be combined to generate specialized and highly optimized systems fulfilling specific application requirements. The goals of GP are to (1) decrease the conceptual gap between program code and domain concepts (known as achieving high intentionality), (2) achieve high reusability and adaptability, (3) simplify managing many variants of a component, and (4) increase efficiency (both in space and execution time). GP typically concentrates on single classes that can be parameterized to achieve the required functionality. Despite the powerful customization mechanisms, GP approach generally still remain at the level of abstraction supported by third-generation programming languages. In contrast, our approach focused on higher-level building blocks, such as patterns and domain-specific languages, which can be instantiated similar to the way that templates are parameterized in GP. Role-based descriptions of the solution could thus be treated as a type of template that spans multiple classes.

Aspect-oriented software development (AOSD) is a GP technology designed to more explicitly separate concerns in software development. AOSD techniques [13] make it possible to modularize crosscutting aspects of complex distributed systems. An aspect is a piece of code or higher-level construct, such as implementation artifacts captured in an MDA platform-specific model (PSM), which describes a recurring property of a program that crosscuts the software application. In our approach, a role-based solution could represent either a crosscutting concern or a concern that could be modularized using OO techniques. In the case of crosscutting concerns, we need to implement model transformation to distribute the particular functionality over the application logic. This task is similar to the task typically performed by weavers in AOP.

Scope, Commonality, and Variability (SCV) analysis [24] is related work on domain engineering that focuses on identifying common and variable properties of an application domain. SCV uses this information to guide decisions about where and how to address possible variability and where the more "static" implementation strategies could be used. Our approach supports SCV and makes it possible to capture commonality and variability at a level that is closer to the problem domain compared with third-generation programming languages. In addition, pattern feature inheritance provides a powerful mechanism to deal with variability at higher abstraction levels by enabling the substitution of pattern-based system building blocks, similar to the substitution at the class supported provided by inheritance in OO design and programming.

In [18] the authors describe a role-based approach to forward and reverse-engineering to introduce or find pattern instances in existing code. Their approach is similar to what we suggesting in this paper. The main difference is that the feature inheritance relationships between patterns that we propose are designed to allow better substitution and composition at the model level.

## 5. Concluding Remarks

This paper presents the novel approach to pattern classification and composition by introducing the feature inheritance relationships between patterns. We also demonstrate how patterns can be used as higher-level building blocks to support the introduction of new aspects without affecting the main application logic. This approach is possible because of relationships between patterns that are analogous with inheritance in OO programming languages.

The work described in this paper provides the conceptual foundation for a certain type of model transformation that preserves key properties of applications being developed. This type of transformation can be treated as a symmetrical transformation and used to allow better substitutability of model parts defined as role-based solution templates. Our work also enables the automation of role-mapping process by MDD tools based on feature inheritance relationships between patterns. Pattern feature inheritance is an example of symmetrical transformation that is important for the next generation of modeling tools, which need to manipulate higher-level building blocks, such patterns or other role-based solutions.

The ultimate goal of our work is to create an Integrated Concern Manipulation Environment (ICME) [2][3], which is an MDD toolsuite that allows manipulation (i.e., adding, removing, and specializing) different aspects of large-scale distributed software systems using higher level building blocks (such as patterns and aspect-oriented techniques) to merge these blocks unobtrusively with the application logic implementations. To provide such ICME manipulation functionality we need to determine how to formalize pattern composition rules. In the pattern literature, *forces*,

*benefits*, and *liabilities* are mentioned as key factors to make decisions about which pattern to use in which contexts and how to combine patterns together effectively. Our future work will analyze these descriptions in various patterns and devise MDD-based formalisms and tools that support automated and/or semi-automated analysis of pattern usage and composability. For example, MDD wizards can guide users through decision processes by asking questions and navigating through a graph of patterns to select suitable patterns.

## References


[1] OMG: "*Model Driven Architecture (MDA)*" Document number ormsc/2001-07-01 Architecture Board ORMSC1, July 9, 2001.

[2] A. Nechypurenko, T. Lu, G. Deng, D. C. Schmidt, A. Gokhale. "*Applying MDA and Component Middleware to Large-scale Distributed Systems: A Case Study*," Proceedings of First European Workshop on Model Driven Architecture with Emphasis on Industrial Application, University of Twente, Enschede, The Netherlands, 2004.

[3] A. Nechypurenko, T. Lu, G. Deng, E. Turkay, D. C. Schmidt, A. Gokhale. "*Concern-based Composition and Reuse of Distributed Systems*." Proceedings of the 8th International Conference on Software Reuse, 2004.

[4] D. C. Schmidt, S. D. Huston. *C++ Network Programming: Mastering Complexity Using ACE and Patterns*. Addison-Wesley Longman, 2003.

[5] D. C. Schmidt, S. D. Huston. *C++ Network Programming: Systematic Reuse with ACE and Frameworks*. Addison-Wesley Longman, 2003.

[6] P. Tarr, H. Ossher, W. Harrison,S.M. Sutton Jr., "*N Degrees of Separation: Multidimensional Separation of Concerns*," Proceedings of the 21st International Conference on Software Engineering, ACM, New York, 1999, pp. 107--119.

[7] B. Liskov, "*Data Abstraction and Hierarchy*". SIGPLAN Notices, 23,5, May 1988, p. 25.

[8] D. M. Weiss. "*Defining Families: The Commonality Analysis*," Proceedings of the 21st International Conference on Software Engineering, Los Angeles, 1999, pp. 671 - 672.

[9] M. Mezini and K. Ostermann. *"Integrating Independent Components with On-Demand Remodularization*," Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOSPLA), Seattle, Washington, USA, November 4-8, 2002.

[10] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, "*Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects"*, Volume 2, Wiley & Sons, New York, 2000.

[11] J. Gray, J. Sztipanovits, T. Bapty Sandeep Neema, A. Gokhale, and D. C. Schmidt, "*Two-level Aspect Weaving to Support Evolution of Model-Based Software*," Aspect-Oriented Software Development, Edited by Robert Filman, Tzilla Elrad, Mehmet Aksit, and Siobhan Clarke, Addison-Wesley, 2003.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. "*Design Patterns: Elements of Reusable Object-Oriented Software*." Addison Wesley, 1995.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. "*Aspect-oriented programming*", Proceedings of ECOOP'97, Jyvaskyla, Finland, 1997.

[14] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, "*Pattern-Oriented Software Architecture—A System of Patterns"*, John Wiley and Sons, 1996

[15] D. C. Schmidt, D. L. Levine, and S. Mungee, "*The Design and Performance of Real-Time Object Request Brokers*" *Computer Communications*, vol. 21, no. 4, pp. 294–324, Apr. 1998.

[16] D. C. Schmidt et. al, "*TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems*", IEEE Distributed Systems Online, vol. 3, no. 2, Feb. 2002.

[17] M. Völter, A. Schmid, E. Wolff. *Server Component Patterns: Component Infra-structures Illustrated with EJB*, Wiley and Sons, 2002.

[18] Gert Florijn, Marco Meijers, and Pieter van Winsen, "*Tool Support for Object-Oriented Patterns*" Proceedings of ECOOP'97, Jyvaskyla, Finland, 1997.

[19] J. G. Siek, L. Lee, A. Lumsdaine. "*Boost Graph Library, the User Guide and Reference Manual"*. Addison Wesley.

[20] "eXecutable UML (xUML)," Kennedy Carter, http://www.kc.com.

[21] George T. Heineman and Bill T. Councill, "*Component-Based Software Engineering: Putting the Pieces Together"*, Addison-Wesley, Reading, Massachusetts, 2001.

[22] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent, "*Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools"*, John Wiley & Sons, New York, 2004.

[23] Krzysztof Czarnecki, Ulrich Eisenecker. "*Generative Programming: Methods, Tools, and Applications"*. Addison-Wesley Pub Co.

[24] J. Coplien, D. Hoffman, D. Weiss, "*Commonality and Variability in Software Engineering*", *IEEE Software*, November/December 1999, pp. 37-45.

[25] I. Jacobson, G. Booch, J. Rumbaugh. "*The Unified Software Development Process"*. Addison-Wesley Professional, 1999.

[26] J. Coplien and L. Zhao. "Symmetry Breaking in Software Patterns," Springer Lecture Notes in Computer Science Series. , 2001.

[27] J. Coplien. "The Future of Language: Symmetry or Broken Symmetry?" Proceedings of VS Live 2001, San Francisco, California, January 2001.

[28] Priya Narasimhan, Louise E. Moser, and P. M. Melliar-Smith, "*Using Interceptors to Enhance CORBA,*" IEEE Computer, July 1999.

[29] J. Rosen, "Symmetry in Science: An Introduction to the General Theory," pp 9-10. New York: Springer-Verlag, 1995.

[30] Steve Berczuk, A Pattern for Separating Assembly and Processing, *Pattern Languages of Program Design*: *Volume 1*, Addison-Wesley, 1995.