# Towards the Use of Dependencies to Manage Variability in Software Product Lines

Roberto Silveira Silva Filho, David F. Redmiles
Donald Bren School of information and Computer Sciences
Department of Informatics, University of California, Irvine
Irvine, CA 92697-3430 USA
{rsilvafi, redmiles}@ics.uci.edu

***Abstract***

*Dependencies have been used in feature-oriented product line to manage feature incompatibilities alternatives, activation requirements, and to support the built of different software configurations. Few studies, however, have been devoted to study the role of dependencies in limiting the variability of product lines and as important criteria for selecting variability realization techniques. Understanding those variability implications, allow us to better understand the design trade-offs of a particular product line, to bound its variability dimensions, and to decide, as early as in the design phase, where and which variability realization techniques to apply. This position paper proposes the use of dependencies as one of the main criteria to be used in bounding variability and choosing the appropriate variability realization techniques. We motivate and exemplify our approach with a publish/subscribe product line.*

## 1 Introduction

The goal of software product lines is "to capitalize on commonality and manage variability in order to reduce the time, effort, cost and complexity of creating and maintaining a product line of similar software systems"[1] Whereas in a software product line, reuse allows the reduction of the costs of producing similar software systems, variability permits the customization of a software family to the different needs of the product line members. It also facilitates the incorporation of new software products in a product family, thus adding value to the product line (Baldwin and Clark 2000). Variability, however, comes with a cost. The more variability a product line supports, the more complex its implementation becomes. Dependencies from the problem domain and from the variable feature set pose a limit in the solution variability by hindering the reuse of existing features, increasing the solution complexity and posing extra burden in the configuration strategies.

In the design of software product lines, feature-oriented approaches have been successfully used in the industry, where there seems to be a consensus on the use of feature-oriented design models. In this approach, the variability among software products is modeled in terms of features. Features represent units of variation in different versions of the software (Svahnberg, Gurp et al. 2005). Ideally, features must be implemented (or realized) as independent (or modular) pieces of code that can be specified and built in isolation from one another. In practice, however, features are not completely independent; those units of variability usually require different services from other features either through hierarchical decomposition or by all sorts of "use" dependencies. According to the complexity of the system, this hierarchy can have an arbitrary number of levels. Moreover, due to variability and domain constraints, incompatibilities among features may also exist. Hence, in a feature-oriented model, the representation of features dependencies is crucial. Features may be incompatible (exclude dependency), may require additional or complementary features (usage dependency), may modify the behavior of other features (modify dependency), and may also have special activation dependencies (multiple, exclusive, subordinate, concurrent or sequential) (Lee and Kang 2004). Additionally, due to reuse and the variability realization technique employed, different features may share common sub-features. Those dependencies can impact the design and behavior of software in different ways. For example, they can originate unforeseeable behavior in software as the case of feature interference problem (Cameron and Velthuijsen 1993) where the combination of two or more features that share common resources can interfere with one another in unpredictable ways.

With such a variety of dependencies and relations between features, the choice of the appropriate variability realization technique (such as whether to employ component or aspectual decomposition; or to apply compile time or runtime variability for example) may be the difference between a tangled and a modular implementation, guaranteeing important characteristics to the software product line such as maintainability, modularity, comprehensibility and the extensibility. Moreover, the understanding of those dependencies and their impact in the system properties may support variability bounding and design simplifications, allowing designers to better assess the costs of varying or fixing certain aspects of the product line.

[1] Extracted from www.softwareproductlines.com/introduction/concepts.html

# 2 Background and Motivation

The study of the role of dependencies in software product lines has gained recent attention from the research community. The focus, however, has been more on the use of those dependencies to prevent architecture configuration mismatch, and less on the study of the impact of those dependencies on the system design complexity and their impact on the variability of software. For example, in the FODA (Kang, Cohen et al. 1990), FORM (Kang 1998), RSEB (Griss, Favaro et al. 1998) methods and in the generative approach in (Czarnecki and Eisenecker 2000), dependencies are used to model usage interactions (alternative, multiple, optional and mandatory) as well as incompatibility relations (exclusive or excludes), with the focus on configuration management and conflict resolution. Recently, (Ferber, Haag et al. 2002) stresses the importance of dependency analysis in feature diagrams, and proposes a separate feature-dependency model that complements the existing feature tress. Additionally, it characterizes different interactions between features such as intentional, environmental, and usage dependencies. Finally, in a more recent work, (Lee and Kang 2004) studied the role of dependencies on modeling runtime feature interactions, introducing the notion of activation and modification dependencies in feature diagrams.

In the implementation domain, feature dependencies usually manifest themselves in the form of coupling between the components that implement those features, in special data and control coupling occur as a consequence of activation and usage dependencies. Those dependencies have different impacts in the variability of the final solution. Whereas control coupling usually limits the activation order of the different pieces of software, data coupling can limit the variability and reuse of those components. (Parnas 1978; Stevens, Myers et al. 1999)

On the light of those problems, different variability realization approaches have been used. For example, (Lee and Kang 2004) propose a set of object-oriented realization strategies to address activation dependencies. Those strategies are presented in the form of design patterns derived from existing Factory, Proxy and Builder patterns (Gamma, Helm et al. 1995). In essence, those patterns focus on managing and enforcing activation dependencies by promoting the late-binding of the components that implement the many software features. Whereas useful in many contexts, this modular (object-oriented) decomposition is not always sufficient to address other kinds of dependencies, especially crosscutting variability dimensions or aspects, originated from more fundamental problem dependencies. This motivated recent work such as (Garcia, Sant'Anna et al. 2005), where Aspects are used to modularize design patterns.

In this position paper, we argue towards a more deep understanding of the role in dependencies in software product lines. Not only as important information for configuration management support, but as main factors to be considering in the design, bounding and variability realization selection phases. We exemplify the role of dependencies with the following case study.

## 2.1 Case Study: Publish/subscribe Product Line

Pub/sub infrastructures provides an asynchronous message service where information providers (publishers) generate information in the form of events (or messages); whereas information consumers (or subscribers) express interest in those events by means of subscriptions. Based on the subscription (an expression or query that can include the event content, order or time restrictions), the events are routed from the publishers to the appropriate subscribers. The events are then delivered according to a notification policy. Different publish/subscribe systems have been built from scratch in the last years, being tailored to different application domains. This observation motivated our research in the development of YANCEES (Silva Filho and Redmiles 2005), a flexible infrastructure, that can be tailored to the needs of different publish/subscribe application domains.

One important step in the design of a product line is the problem domain analysis and the identification of the main units of variability (the features). In our design, we adopted the framework proposed by (Rosenblum and Wolf 1997). In this model, routing, event, notification, observation (or subscription), timing and resource are the main design concerns of a publish/subscribe infrastructure. They represent the main variability dimensions in our model. The event model defines how the event is represented (for example: tuples, record, object or plain text). The routing model defines the strategy used to match subscriptions to events (whether by the content, by a specific field (topic), or by a dedicated channel where all events produced are delivered to the subscriber). The notification model defines how to deliver events to the subscribers once they are matched with the subscription (push or pull). The subscription model defines the query language, and all the commands that can be part of it. Those commands may operate over the content or over the order of the events. The timing model defines guarantees with respect to the total or partial order of the events. The resource model specifies how the infrastructure is implemented (whether distributed in a peer-to-peer or hierarchical fashion, or whether it is centralized). Note that the abstract features (routing, event, notification, subscription, timing and resource), define a set of variability dimensions that are further specialized in the following hierarchy level by different optional features.

A possible feature diagram representing such product line variability is depicted in Figure 1. The Diagram uses a UML notation. Stereotypes are used to express optionally (OR relation) and exclusivity (XOR relation). An optional feature can be selected together with other optional features in the same level, for the same super feature. Abstract features appear as the first level under the pub/sub infrastructure concept, and are not marked with stereotypes. Aggregation indicates containment and

composition implies a part-role relation of the pub/sub concept. When no stereotype is used, the features or concepts are mandatory.

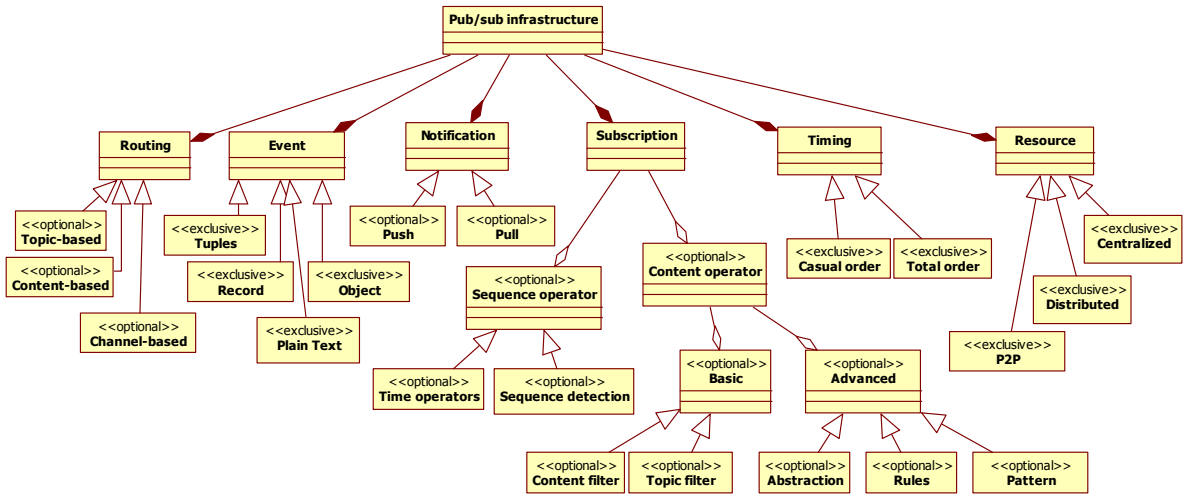


**Figure 1 Feature Diagram of Pub/sub systems**

## 2.2 The Role of Fundamental and Configuration-specific Dependencies

Feature diagrams, as the one presented in Figure 1, express the basic model of feature-oriented product line design. A feature diagram represents a tree of features where the root represents a concept. A concept is usually represented by a set of mandatory, optional and abstract features. The first level of a feature diagram usually represents a set of abstract features that implement a concept, in our example, a pub/sub infrastructure. Those abstract features are parents of more specific optional, alternative, exclusive and mandatory features. For example, routing, subscription, notification, event, timing and resource features, which variability is further defined by means of concrete features such as record-based for event, total order timing, content-based filtering for subscription and so on.

In a feature-based approach, we define as **fundamental dependencies**, those relations between abstract features that are imposed by the problem domain. In other words, they comprise the interactions between generic abstract features that define the problem domain. In our example, events, routing, subscriptions and notifications are inter-related by the very publish/subscribe process itself: events are routed following subscriptions, generating notifications. This process is common to all publish/subscribe infrastructures and provides a conceptual model where configuration-specific features can be "plugged in"; moreover, they define our variability dimensions in the product line. In contrast to fundamental dependencies, **configuration-specific dependencies** are those dependencies that exist between optional features and/or the components that implement them. For example, some notification servers may use pull notification approach, which require event persistency and user authentication support, sub-features present only in certain members of the product family, specializations of the generic notification feature.

In our case study, as seen in Figure 1, variability exists in every design dimension of the system. A problem then surfaces when those variability dimensions (or abstract features) are further refined and implemented. First, the dependencies between features are not easily visualized in the diagram; they in fact are not represented. Second, the abstract features are usually implemented as part of the base or common code, whereas the optional features are provided on specific configurations. As a consequence, the abstract pub/sub fundamental dependencies implicitly impact the implementation of each feature of the model. For example, the implementation of a subscription command in our case study must account for the way events are represented (records, objects, plain text, attribute-value pairs) and the routing guarantees that the infrastructure provides (total or partial order of events). Those implicit dependency usually become encoded in the base implementation of the product line, and will manifest themselves later in the implementation of the infrastructure. As a consequence, every time the event format or the routing policies vary, the subscription commands that depend on those parameters will need to vary, either by providing alternative implementations, or by accounting for this variability in the feature implementation itself.

This observation has important consequences: (1) **the combinatorial explosion of features**. Feature dependencies work as an important variability limitation factor for the product line: the more dependencies exist between features, the harder it is to manage all possible combinations of features in a product line. (2) **Reduced generality of features**: dependencies also limit the reuse of existing features and their implementations since changes in one feature can impact features that depend on it. This fact prevents the unbounded generalization of product lines, and places a theoretical limit in how one can leverage reuse

in such systems. (3) **Limited extensibility**: Since new features need to cope with the existing dependencies in the model their implementation tends to be more complex and prone to errors. Using Parnas terminology (Parnas 1976), the dependencies from the incomplete program (or base code), impact the variability of the product family as a whole since implicit domain relations (or dependencies) are usually inherited from the incomplete code.

As a consequence, in the design of a product line, a balance between variability and reuse needs to be achieved. In this position paper, we argue that the dependencies represent the key to understand and tackle this problem. With such information, designers can either limit or fix the variability of a certain dimension or choose a variability realization technique that minimizes such coupling. In spite of its importance, both fundamental and configuration-specific dependencies are usually not explicitly represented in feature diagrams. Instead of augmenting existing diagrams with such information, we propose the use of alternative diagrams and models to elucidate those dependencies (Kruchten 1995).

# 3 Approach

In our work, we propose the study of dependencies in a more fundamental way, understanding the implications of those feature relationships in the choice of the most adequate variability realization techniques. Our approach comes from the observation that abstract problem features, originated in the problem domain are usually chosen as main variability dimensions in the problem. Those features, however, have implicit fundamental dependencies (or coupling) that becomes part of the common code of the architecture and ends up constraining the variability of the product line as a whole, and the features specifically. Hence, the analysis of those fundamental dependencies can help us select the proper combination of variability dimensions and feature realization approaches in order to reduce their impact in the software that will ultimately implement the model. Also, depending on the target software and hardware platforms or different environmental constraints and limitations, the dependency model can provide a basis to decide which variability dimensions to fix, and/or which ones to make variable. The approach can be summarized as:

1. Perform initial feature domain analysis
2. Identify the abstract features in the domain that define an abstract system commonality
3. Perform a dependency analysis with respect to the abstract feature dependencies
4. Consider the target platform hardware and software constrains
5. Bound the mandatory features variability in order to minimize the impact of dependencies
6. Choose appropriate variability realization technique for the base code
7. Choose the variability realization technique for the optional code

The variability realization approach used to model a feature will impact the variability of the system in different ways. For example, (Czarnecki and Eisenecker 2000) defines two main decomposition approaches: Modular decomposition and Aspectural decompositions. When used in conjunction, those approaches can complement one another and reduce the impact of dependencies (or coupling) in the code. Those approaches are discussed in mode detail in the next sections.

## 3.1 Making dependencies explicit

The feature diagram in Figure 1 does not allow the visualization of all the dependencies between the problem features, depicting only optionally, aggregation, specialization and exclusion. One way to visualize dependencies is proposed by (Ferber, Haag et al. 2002). The use of graphs to represent such information, however, does not scale. Instead, we propose the use the DSM (Dependency Structure Matrix) notation as that used by (Baldwin and Clark 2000). In this representation, dependencies between parameters, or in our case features, are represented in the form of a square matrix. In our approach, instead of representing only the number of dependencies or their simple existence, with an X for example, we label the dependencies with the kind of coupling they provide. A ‘D’ is placed in the matrix to represent a data dependency if the $i^{th}$ column depends on the dimension in the $j^{th}$ row; similarly, a ‘C’ is placed o represent a control dependency. The DSM of our case study is presented at Figure 2.

Figure 2 reveals some interesting dependencies between the main variability dimensions of the product line. Note that the event model and its representation directly impacts the subscription and routing models. A change in the event format requires a change in the subscription language and routing algorithms due to a strong data coupling between those two features or dimensions (the event content and format itself). Timing is another crucial feature in the model. A change in the routing algorithm may impact the timing guarantees of the product line (guaranteed delivery and total order of events), which will impact the subscription language semantics. A change in the resource model may also affect the timing model. For example, in a hierarchical distributed system, the total order of events may not be available. Finally, the notification model is orthogonal to the other features. Since it manages only events, it can vary independently form the other features. Hence, this simple analysis allows us to draw two lessons: first, by analyzing the dependencies between the abstract features, as in Figure 2, a system architect can identify relations that are not initially obvious in the original feature model of the system; and second, as

a consequence, she can adopt some strategies to minimize the impact of dependencies in the final variability of the product line. For example, use different decompositions such as aspects, or even fix some variability dimensions, such as the event model. In doing so, the design of a product line can be optimized and pitfalls such as hidden dependencies can be assessed, managed or limited.

| | | A | A1 | A2 | B | B1 | B1.1 | B1.2 | B2 | B2.1 | B2.2 | B2.3 | B2.4 | B2.5 | C | C1 | C2 | C3 | D | D1 | D2 | D3 | D4 | E | E1 | E2 | F | F1 | F2 | F3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **A** | **notification** | . | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **A1** | _push | | . | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **A2** | _pull | | | . | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **B** | **subscription** | | | | . | | | | | | | | | | | | | | | | | | | | | | | | | |
| **B1** | **_sequence** | | | | | . | | | | | | | | | | | | | | | | | | | | | | | | |
| **B1.1** | __order | | | | | | . | | | | | | | | | | | | | | | | | | | | | | | |
| **B1.2** | __interval | | | | | | | . | | | | | | | | | | | | | | | | | | | | | | |
| **B2** | **_content** | | | | | | | | . | | | | | | | | | | | | | | | | | | | | | |
| **B2.1** | __filterContent | | | | | | | | | . | | | | | | | | | | | | | | | | | | | | |
| **B2.2** | __filterTopic | | | | | | | | | | . | | | | | | | | | | | | | | | | | | | |
| **B2.3** | __abstraction | | | | | | | | | | | . | | | | | | | | | | | | | | | | | | |
| **B2.4** | __rules | | | | | | | | | | | | . | | | | | | | | | | | | | | | | | |
| **B2.5** | __pattern | | | | | | | | | | | | | . | | | | | | | | | | | | | | | | |
| **C** | **routing** | | | | C | | | | | | | | | | . | | | | | | | | | | | | | | | |
| **C1** | _topic | | | | C | | | | | | | | | | | . | | | | | | | | | | | | | | |
| **C2** | _content | | | | C | | | | | | | | | | | | . | | | | | | | | | | | | | |
| **C3** | _chanel | | | | C | | | | | | | | | | | | | . | | | | | | | | | | | | |
| **D** | **event** | | | | D | | | | D | D | D | D | D | D | D | D | D | | . | | | | | | | | | | | |
| **D1** | _tuple | | | | D | | | | D | D | D | D | D | D | D | D | D | | | . | | | | | | | | | | |
| **D2** | _record | | | | D | | | | D | D | D | D | D | D | D | D | D | | | | . | | | | | | | | | |
| **D3** | _text | | | | D | | | | D | D | D | D | D | D | D | D | D | | | | | . | | | | | | | | |
| **D4** | _object | | | | D | | | | D | D | D | D | D | D | D | D | D | | | | | | . | | | | | | | |
| **E** | **timing** | | | | C | C | C | C | | | | | | | C | C | C | | | | | | | . | | | | | | |
| **E1** | __casualOrder | | | | C | C | C | | | | | | | | C | C | C | | | | | | | | . | | | | | |
| **E2** | __totalOrder | | | | C | C | C | | | | | | | | C | C | C | | | | | | | | | . | | | | |
| **F** | **resource** | | | | | | | | | | | | | | | | | | | | | | | C | C | C | . | | | |
| **F1** | __federated | | | | | | | | | | | | | | | | | | | | | | | C | C | C | | . | | |
| **F2** | __centralized | | | | | | | | | | | | | | | | | | | | | | | C | C | C | | | . | |
| **F3** | __P2P | | | | | | | | | | | | | | | | | | | | | | | C | C | C | | | | . |

**Figure 2 DMS showing the dependencies between the features.**

## 3.2 Variability realization techniques

**Modular decomposition** aims at decomposing the system into a hierarchy of modules (components, objects, methods and so on) in such a way that cohesion in the modules are maximized, while coupling is minimized. Many recurring solutions exist to help in the design of such systems, including design patterns (Lee and Kang 2004), conditional compilation, templates and other approaches(Svahnberg, Gurp et al. 2005) that are usually designed for functional or object-oriented programming.

Modular decomposition is not always possible to accomplish in object-oriented languages due to what is called crosscutting concerns, that can be a result of non-functional requirements for example. Also, due to the lack of modularity in many object-oriented design patterns (Hannemann and Kiczales 2002), their use throughout the system, and especially in the base code, makes it hard for the software to support changes and support evolution. For each design pattern applied, new dependencies are introduced to the product line architecture, as well as additional configuration costs to manage those dependencies.

Moreover, it is usually the case that a feature is mapped not to a single component but to a set of components installed in different parts of the base code. In our example, this approach can be applied in the case of Notification and Resource models in Figure 2. Those models do not depend on any other model.

**Aspectural decomposition** aims at decomposing a system into a set of perspectives. Each one of those perspectives comprises concerns that refer to a common model. Another way of describing this approach is that aspects encapsulate the coupling that might exist between components that implement a single feature, that otherwise would usually become hard coded in the many components that implement a feature. In the example of Figure 2, control dependencies can be modularized as aspects, that weave to the base code, the appropriate algorithm, according to the selected timing constraints. Examples of modularizations using aspects are discussed at (Garcia, Sant'Anna et al. 2005) and (Hannemann and Kiczales 2002).

Other decomposition and strategies are also possible. In the pub/sub example, data coupling can be addressed by using reflection as described in (Eugster and Guerraoui 2001).

## 3.3 Bounding and restrictions

In order to reduce the coupling between dimensions, one simple alternative is the bounding of variability dimensions. After a dependency analysis, designers can opt for limiting the variability of some abstract features. For example, the event dimension in Figure 1 can be restricted to support only attribute-value representation. This representation is generic enough to be used as topic and object-based representations by using some conventions. The textual event representation, however, is not trivially supported in this model, and must be adapted to conform to attribute-value representations. A trade-off, therefore, between variability and bounding exists and needs to be considered, according to the applications the system will support.

Environmental restrictions also play an important role. Variability techniques such as aspectual decomposition or reflection may not be available in a given platform. In this case, whenever bounding is not an option, more traditional approaches such as design patterns (Lee and Kang 2004) may be used.

# 4 Final Considerations

Feature dependencies are important in product line design since they can limit the variability of the software. There are two major categories of dependencies: fundamental problem dependencies, coming from the common problem features, that originate the common base code; and feature-specific dependencies that are originate from optional and alternative features in the product line. The nature of the dependencies between the product line features, especially those that are part of the problem domain, have a deep impact in the resulting variability of the software solution, a consequence of the coupling of the components in the final code implementation. The understandings of those variability dimensions, allow us to analyze the design trade-offs of a particular product line, and must be considered in the design of product lines. The use of dependency analysis tools such as DSMs can help designers identify, as early as in the design phase, the dependencies between the main variability dimensions of the problem. This information is crucial to support designers in choosing the appropriate variability realization techniques that can better suit the implementation (or realization) of the features, or to select which variability dimension to fix in order to simplify the architecture design.

Whereas dependencies alone are not the only criteria to be used in selecting a variability realization approach, their understanding provides an important input to practitioners in understanding the variability restrictions imposed by the problem domain, helping them in the choice of the adequate technique for their case.

# Acknowledgements.

This research was supported by the U.S. National Science Foundation under grants 0534775, 0326105, and 0205724 and by the Intel Corporation.

# References

Baldwin, C. Y. and K. B. Clark (2000). Design Rules, Vol. 1: The Power of Modularity. Cambridge, MA, MIT Press.

Cameron, E. J. and H. Velthuijsen (1993). "Feature interactions in telecommunications systems." IEEE Communications Magazine **31**(8): 18-23.

Czarnecki, K. and U. W. Eisenecker (2000). Generative Programming - Methods, Tools, and Applications, Addison-Wesley.

Eugster, P. T. and R. Guerraoui (2001). Content-Based Publish/Subscribe with Structural Reflection. 6th USENIX Conference on Object-Oriented Technologies and Systems, COOTS'01, San Antonio, TX.

Ferber, S., J. Haag, et al. (2002). "Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line." Lecture Notes in Computer Science. Second International Conference on Software Product Lines, SPLC'02 **2379**: 235-256.

Gamma, E., R. Helm, et al. (1995). Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company.

Garcia, A., C. Sant'Anna, et al. (2005). Modularizing design patterns with aspects: a quantitative study. Aspect-oriented software development, Chicago, Illinois, ACM Press.

Griss, M. L., J. Favaro, et al. (1998). Integrating Feature Modeling with RSEB. Fifth International Conference on Software Reuse.

Hannemann, J. and G. Kiczales (2002). Design Pattern Implementation in Java and AspectJ. 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02), Seattle, Washington.

Kang, K. C. (1998). "FORM: A Feature-Oriented Reuse Method with Domain Specific Architectures." Annals of Software Engineering **5**: 345-355.

Kang, K. C., S. G. Cohen, et al. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study - CMU/SEI-90-TR-021. Pittsburgh, PA, Carnegie Mellon Software Engineering Institute.

Kruchten, P. B. (1995). "The 4+1 View Model of architecture." IEEE Software **12**(6): 42-50.

Lee, K. and K. C. Kang (2004). "Feature Dependency Analysis for Product Line Component Design." Lecture Notes in Computer Science - 8th International Conference on Software Reuse, ICSR'04 **3107**: 69-85.

Parnas, D. L. (1976). "On the Design and Development of Program Families." IEEE Transactions on Software Engineering **SE-2**(1): 1-9.

Parnas, D. L. (1978). Designing software for ease of extension and contraction. 3rd international conference on Software engineering, Atlanta, Georgia, USA, IEEE Press.

Rosenblum, D. S. and A. L. Wolf (1997). A Design Framework for Internet-Scale Event Observation and Notification. 6th European Software Engineering Conference/5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Zurich, Switzerland, Springer-Verlag.

Silva Filho, R. S. and D. Redmiles (2005). Striving for Versatility in Publish/Subscribe Infrastructures. 5th International Workshop on Software Engineering and Middleware (SEM'2005), Lisbon, Portugal., ACM Press.

Stevens, W. P., G. J. Myers, et al. (1999). "Structured design." IBM Systems Journal **38**(2-3): 231 - 256.

Svahnberg, M., J. v. Gurp, et al. (2005). "A Taxonomy of Variability Realization Techniques." Software Practice and Experience **35**(8): 705-754.