Product-Line Architectures

Martin L. Griss, Ph.D.
Laboratory Scientist
Software Technology Laboratory
Hewlett-Packard Company, Laboratories
Palo Alto, CA, USA

(*Chapter 22 in* ***Component-Based Software Engineering: Putting the Pieces Together,*** *Edited by George T. Heineman, Ph.D. & William Councill, M.S., J.D., May 2001, Addison-Wesley*

1. Introduction

A *product-line* is a set of products that share a common set of requirements, but also exhibit significant variability in requirements. There are an increasing number of organizations that understand the business importance of managing related products as members of a product-line, and several of these organizations also understand how component reuse can help.

You can manage a set of products as a group, planning the whole set consistently, allocating funding and developers to several of the products at the same time and advertising the products as a set, highlighting their common and varying features. This commonality and variability can be also exploited during development by treating the set of products as a family and decomposing the software into shared, reusable components that separate concerns (Parnas 1976, 1979). For example, at Hewlett-Packard, several families of LaserLet printers are developed, managed and sold as product lines. Different models (color or black and white, personal or business, large or small) provide different capabilities, and product numbers , features and web-pages (www.hp.com) are arranged to make comparisons of commonalties and differences easy. In many cases, several members of each product line share some common hardware, firmware and software components, but also differ in other components. For example, larger color LaserJets have complex paper handling systems, while smaller printers do not.

Success with product-line development based on reusable components requires orchestrated attention to a variety of both technical and non-technical issues, including how to design the components and the products to maximize product-line reuse, what component technology to use, defining a component-based product-line development process, and a host of issues related to organization structure, roles and culture, adoption strategies and business strategy. Component reuse can play a significant role in reducing costs, decreasing schedule time, and ensuring commonality of features across the product-line. Business managers can make strategic investments in creating and evolving components that benefit a whole product-line, not just a single product. Common components are reused multiple times, and defect repairs and enhancements to one product can be rapidly propagated to other members of the product-line.

In this chapter, I will describe an approach to systematic, architected product-line development: analysis, design, implementation and customization of reusable components that is the essence of product-line CBSE. As in chapter 9, I find it useful to highlight the architectural phase of product and component design – architecture is about the overall structure, subsystems, key interfaces and key mechanisms. In particular, when considering product-line development, I find it helpful to think first about the architecture of the individual products, and the common architecture that underlies or shapes the entire product-line. Then I consider the components and infrastructure that implement this common core, essentially a skeletal product or so-called "framework." You must carefully design and structure the components and framework to express, leverage and manage the customizable properties of components.

Recent, largely independent work in the mostly separate reuse, object-oriented and architecture communities has reached the point where I see that the integration of the activities promises to yield a new coherent approach to product-line development. This approach starts from a systematic analysis of the set of common and variable features supporting a product-line, then defines a set of reusable elements, which can then be customized and combined into components and frameworks to implement the products. More details on how this integration might proceed can be found in a series of papers by the author and colleagues (Griss, Favaro, and d'Alessandro 1998; Griss 2000, 2000a).

1.1 Product-lines Can Exploit Reuse

Product-line CBSE delivers the promise of large-scale software reuse by promoting the use of software components built by commercial vendors or in-house developers. One of the key success factors for CBSE is that the systems produced by the organization form an obvious product-line or application family, and that the organization recognizes a compelling business imperative that can exploit product-line CBSE (see Len Bass, Chapter 21 for more key factors). There is a compelling business need to invest in building and managing a set of products as a family, sharing engineering effort and reusable assets. The driving business reason is the critical need to decrease time to market to meet competitive market forces, reduce overall costs or improve product compatibility. Product-line CBSE then becomes of strategic importance, and management will pay attention.

Product-line CBSE has the potential to:

- Reduce significantly the cost and time-to-market of enterprise software systems by allowing the systems to be built by assembling reusable components rather than from scratch.
- Enhance the reliability of enterprise software systems because each reusable component has gone through several review and inspection stages in the course of its original development and previous use; and because CBSE relies on explicitly defined architecture and interfaces.
- Improve the maintainability of enterprise software systems by allowing new (higher) quality components to replace old ones.
- Enhance the quality of enterprise software systems by allowing application-domain experts to develop components, and software engineers specialized in component-based software development to assemble the components and build enterprise software systems.
- Enhance the agility of organizations in meeting changing market demands by allowing new products to be quickly built by developing only a few new components, and reusing the rest.

1.2 Terminology

In my work on the Reuse-driven Software Engineering Business (RSEB) and extension to FeatuRSEB (Jacobson, Griss, Jonsson, 1997;Griss, Favaro, d'Allesandro, 1998), applications are defined by a set of connected UML models and software, called an *Application System*, constructed by an *Application System Engineering* process. Applications are engineered from customizable components imported from component groups called *component systems*, which interact with each other in a layered, modular architecture. A component system refers to a set of related components that accomplishes some function larger than does a single component. Component systems are engineered by a *Component System Engineering* process, described in detail in the RSEB book (Jacobson, Griss, Jonsson, 1997).

2. Explicit Representation of Features for Product-Line CBSE

When we examine different applications in the same product-line or problem domain, we often compare these applications based on their features. Some features are directly related to application functionality and offered services, perhaps expressed as use cases; other distinguishing features might relate to the application implementation, the window system, style of menus, or operating system. When developing product-line architecture and components for reuse, it is important to understand those features that must be provided by the set of reusable assets (components, infrastructure, models and special tools), and how those features relate.

A *feature* is a product characteristic that users and customers view as important in describing and distinguishing members of the product-line. A feature can be:

- a specific requirement
- a selection among optional or alternative requirements

- related to certain product characteristics, such as functionality, usability, and performance
- related to implementation characteristics, such as size, operating system or computer, or compatibility with certain standards such as CORBA, HL7, or TCP/IP.

Members of a product-line might all exist at one time, or evolve with the product-line. For example, each different member of a word processing product-line might optionally provide an outline mode, spell-checking, grammar correction, or diagrams, may run in Microsoft® Windows™ or Unix, be large or small, support a small number of files or large number files. Some word processors might have a minimal spell checker, while others will have a large spell checker integrated with a grammar checker.

A product-line can be built around the set of reusable components by analyzing the products to determine the common and variable features using a technique called domain analysis. Then you develop a product structure and implementation strategy around a set of reusable components that can be composed to implement several different products.

Some components do not need to be specialized and are easy to reuse "as is" when they match a desired problem. More abstract components can be specialized to express key variability in an appropriate way at well defined "variation points." In some cases the components are simply customized using parameters or inheritance to account for differences in the products that are not expressible by just selecting alternative components; in other cases, the components are substantially modified to create a unique component for a specific product. If we made no provision for variability, we would have to maintain a very large number of "concrete" component systems. A reuser exploits these mechanisms to customize the component to the particular application. This customizability greatly expands the number of applications to which the components can be applied.

Each component captures some subset of features, or allows features to be built up by composing components. For example, each word processor in the word processor product-line might share a common editing component, and optionally one or more components for spell checking, outlining, or other features. These components may be combined to produce a variety of similar word processors, only some of which would be viable products.

2.1 Domain analysis to construct a product-line feature model

Domain analysis (Arango 1994; Griss 1996; Griss, Favaro, d'Allesandro 1998) is a technique to systematically extract features from existing or planned members of a product-line. The domain analyst uses example systems, user needs, domain expertise, and technology trends to identify and characterize elements that are common to all product-line members, and to characterize and express elements that vary among product-line members.

The feature model produced by domain analysis can be represented diagrammatically in several ways. Figure 1 is an example of a tree-diagram of a feature model for a simple online banking domain. Distilled from many sources, it is an organized, selective, and normalized dictionary that becomes the reference for terminology used by component and product developers. The various nodes in this feature diagram show how some features are composed of lower level features (that must always be present), while other features are optional (the small circle) , or correspond to alternatives selected at construction time or dynamically loaded at run time (small clear and solid diamonds, at variation points, vp-feature). Several different domain analysis methods have been developed. They vary in how they identify the domain and represent the features and their relationships. In my work on extending RSEB into FeatuRSEB (Griss, Favaro, d'Allesandro, 1998), I have built on FODA (Kang 1990, 1999). Kang himself has also extended FODA to a product-line method, FORM(Kang 1998).

Domain analysis is used to cluster sets of features to shape the design of a set of components that cover the product-line, and optionally carry this into design and implementation of a product-line architecture and reusable components, infrastructure, and tools that will be used to construct the product-line. Once the variability structure has been modeled, developers can choose the overall decomposition into components and infrastructure, and select an appropriate variability mechanism to produce generic components. For example, you can use a combination of inheritance and templates, or a preprocessor or generator. Finally, the components are designed in detail and implemented.

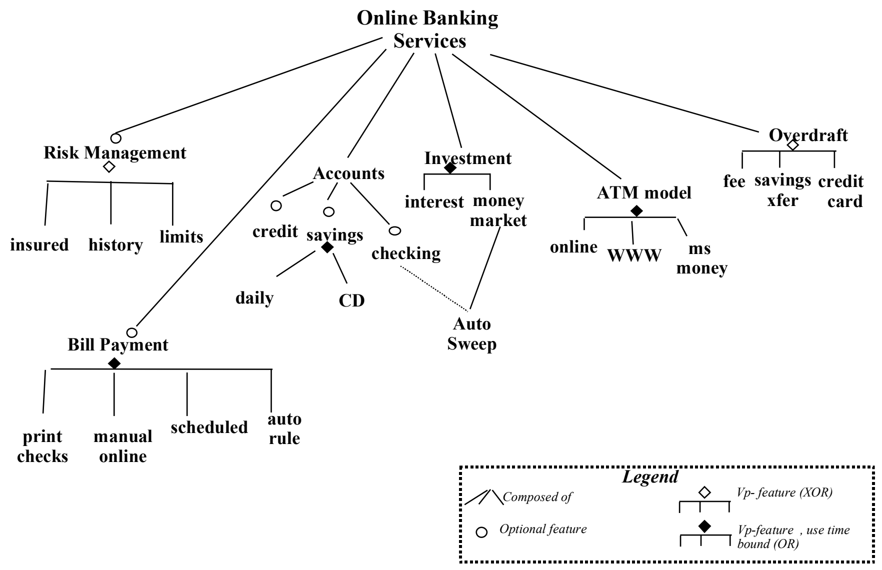

Figure 1 Feature model for a simple online banking domain

In many domains, quite a few of the features are already known by domain experts, which they can use to explain how different kinds of systems differ. In this case, an initial feature model can be constructed without having to go through a formal domain analysis. Often, it will start off looking like the typical class and interface hierarchy that an experienced developer might sketch. But as the domain gets more complex, dependencies and variants will need to be carefully specified using a well-specified and tested method. More details on how UML and FODA are used within FeatuRSEB can be found in (Griss, Favaro, and d'Allesandro 1998).

The feature model is used by product developers to understand which components to select or customize to provide the desired features. When a product developer wants to build a new system in the domain, he consults the feature model to understand what can be combined, selected, and customized in the existing or new system. The feature model also expresses semantic constraints that can't be found in any of the other models—for example, "this operating system can't be selected together with that hardware," and other characteristics that are hard to express directly, such as performance constraints and other dependencies.

The feature model is also used by component developers and product-line architects to determine which components to develop, and how they should be customized, or relate to each other. Different clusterings and decompositions of the set of features into components will lead to different sets of components. For example in the online banking example, we could imagine the Overdraft feature being implemented in a single component, in which the choices of how to implement the overdraft (loan extra money and charge evy fee from account, transfer from savings, or withdraw from credit card) could be selected by a parameter. Alternatively, it may make sense to implement Risk Management and Overdraft together as a related set of components, in which for example, different limits or history would determine what size fee to charge, or which strategy to plug in to a Risk/Overdraft framework.

2.2 Implementing Components Corresponding to Sets of Features

The feature model serves as an index to the other models (such as UML use cases, subsystems, or classes) used during development. A major issue to be addressed is that while some features can be traced fairly directly to specific components, other features trace to component code that is distributed across ("cuts across") multiple components. This relates to the important principle of separation of concerns and reduced component coupling.

Isolating related functionality into one or a small cluster of related components ensures that components can be changed independently of each other, usually without changing their provided or required interfaces. Just as the careful design of component interfaces and the decomposition of a system into relatively independent components is an important part of designing a robust system, so too must we carefully structure the feature model, and decide how to implement the features to corresponding components.

Some features can be implemented as code that is localized to one or a small group of components. For example, features related to language or dictionary choice for “spelling” can be fairly easily expressed by code in the “spelling component.” A change in one of these features (through product evolution, or selection of a variable feature) can be localized to a small number of components.

However, in many other cases, features cut across multiple components, resulting in related code that is distributed across these multiple components. Such “crosscutting” features make it difficult to decompose separate concerns into separate components that hide details of how these features are implemented: for example, security, choice of target operating system or computer, or choice of underlying object model (e.g. COM vs. CORBA). Modifying these “crosscutting” features will have a profound impact on the system. A change in these crosscutting features requires identifying the corresponding fragments of code within multiple methods in multiple components, and making consistent changes. Sometimes a different decomposition into components, using a different architecture, design pattern, or underlying infrastructure, can reduce the crosscutting effect of some features. But, it does not seem possible to do this for all features in all systems.

Managing feature and component evolution becomes extremely complicated when the piece of component code that implements a particular feature or set of closely related features needs to be spread across multiple components, particularly if this code is intertwined with the code corresponding to other features. In some cases, the intertwining can be reduced by an appropriate decomposition into smaller components and well-designed interfaces, but in many cases code contributed by one feature needs to interact fairly closely with that produced by other features. For example, consider the code needed to sort a list or rows of a table. Code to access and compare the elements, and code to traverse and swap list or table elements will be intermixed with the code corresponding to the inner loops of the algorithm, especially for efficiency reasons. Sometimes, the use of macros, inline declarations, templates and abstract data types can be used to allow these different features (choice of element, container, and algorithm) to be manipulated separately, and later combined, as we will describe further below. For example, the C++ Standard Template Library (STL), goes to great length to factor these into separate elements that are then combined into a complete class by the C++ template preprocessor. More powerful techniques will be needed if the features interact across components, especially if these components will need to be in different applications on different computers. For example in comaptible choices of network protocols at both ends of client/server systems.

In general, this problem is mostly related to choosing a particular dominant decomposition into components that does not completely match the dependency and variability patterns of features. In some cases, choosing a different decomposition can improve the robustness for a particular set of features. "Robust" means that for certain kinds of change in requirements and features, the conseqence of the change is localized to only a few components, or locations within a component. Unfortunately, different sets of features “prefer” different decompositions, and it does not seem possible to choose one decomposition that is robust for all changes and evolution (Tarr et al. 1999). The problem is that individual features do not typically trace directly to an individual component or cluster of components -- this means as a product is defined by selecting a group of features, a carefully coordinated and complicated mixture of parts of different components are involved. This "crosscutting" interaction is also referred to as the requirements traceability or feature-interaction problem. In the case of significant crosscutting variability, maintenance and evolution becomes very complicated, since the change in a particular feature may require changes all over the resulting software design and implementation.

I recommend that during use-case design, you should select use cases separately and choose among variant use case features. Each use case typically translates to a collaboration of multiple components, automatically resulting in crosscutting impact. Thus variability in a single use case typically translates into crosscutting variability in a related set of design and implementation components.

2.2.1 Managing crosscutting features

The conventional approach to manage crosscutting features and feature traceability is to build complex maps from features to components and products, and manage the evolution of the features by jointly managing the evolution of the individual components. In the past, this has been done manually, but more recently powerful requirements tracing and management tools (such as DOORS or Calibre RM requirements management tools) have made this somewhat easier. Even with such traceability tools, when one feature changes, code in many components will have to change; to find and decide how to change this code requires developers to disentangle the code corresponding to this feature from the code for other features.

One observation is that a change to a feature (or small sets of related features) is relatively independent from other features, and thus a group of component or product code changes can be related to a few related features. This is similar to the near independence of well designed use cases, described above. Thus, separation of concerns is most effective at the higher-level, before “tangling” and “crosscutting” into design or implementation.

In addition to the use of hyper-web like traceability tools to help manage or find connected pieces of code corresponding to one feature, there are two major approaches to address this problem. Typically, a combination of these is used in practice:

1. ***Architecture*** and ***Component Design*** techniques- use tools and techniques such as patterns (Gamma et al. 1994; Buschmann et al. 1996), interfaces, and styles (Garlan and Shaw 1996), to provide a good decomposition into separate pieces that are relatively independent, localizing changes. This often leads to a better structure and a more maintainable, flexible system. However, discovering and maintaining a good design, requires skill (and some luck) in choosing a decomposition that anticipates changes. Different decompositions are more or less robust against different classes of change.
2. ***Composition or "weaving" of program fragments*** - the feature decomposition is used directly to produce separate code fragments (called "aspects") that are composed or woven together using a language a preprocess or generator to produce complete components or products.

Over the last several years, a number of techniques (“aspect-oriented development”) have been developed to make the "composition and weaving" approach to feature-driven design and development of software practical. An *aspect* is a fragment of code, typically a statement, method or template, perhaps parameterized, that will be composed or woven with base code and other aspects into a complete method, class or component to create complete components and products.

The key idea is to identify a useful class of program fragments that can be mechanically combined using some formal tool, and associating these fragments with features of the software. One does not directly manage the resulting components, only the features and their aspects. One simply dynamically generates a fresh version of the component as needed.

There are several techniques, mechanisms and tools that can be used. These differ primarily in the granularity and language level of fragments and the allowed ways in which they can be combined. For example, is code woven together at the procedure level, at the method level, or within individual statements? Can the approach be seen as a just a disciplined way of using a standard language feature (such as templates or inheritance) or is a new tool added that effectively changes the language (such as a preprocessor or macro)? More detail on several of these techniques can be found in (Griss, 2000, 2000a):

1. Parameterization - parameterized data types, generic programming, parameterized structure inheritance (e.g., frameworks), and parameterized algorithms and/or compile time computation (e.g. for tailoring matrix algorithms and computing primes). In the case of (C++) frameworks, subclasses define new methods that "override" the definition of some default methods inherited from superclasses.
2. Generators - (Bassett, 1996) shows how a frame-based "generator" composes code fragments from several frames. A good decomposition has each frame dealing with one or more aspects. (Batory and O'Malley, 1992; Batory, 1996) developed a specialized preprocessor (GenVoca) that assembles systems from specifications of components, and composition as layers. These techniques have been applied to customizable databases, compilers, data structure libraries and other domains.
3. Subject-oriented programming (SOP) - Ossher, Harrison and colleagues (Harrison and Ossher 1993; Tarr *et al.*, 1999) represent each feature as a separate package, implemented or designed separately. Classes, methods and variables are then independently selected, matched and finally combined together non-invasively using C++ templates to produce the complete program.
4. Aspect-oriented programming (AOP) - developed by Kiczales and colleagues (Kiczales *et al.,* 1997), AOP expands on SOP concepts by providing finer grain support for aspect definition and composition. AOP starts with a base component that cleanly encapsulates some common features in code, using methods and classes. AOP then applies one or more aspects to components to perform large-scale refinements that modify multiple base components. Aspects are implemented using an aspect language (AspectJ ^{TM}) that makes insertions and modifications at defined join points.
5. Templates - the body of a method, or of several methods, is created by composing fragments using C++ templates. For example, in the C++ Standard Template Library (STL), a class was constructed from a composition of an Algorithm, a Container, an Element, and several Adapter aspects (Musser 1996). STL is noteworthy for its lack of inheritance. More recent work has shown that some of the weaving originally thought to need a special generator or aspect language, can be accomplished by clever decomposition into C++ template elements, using a combination of C++ templates and inheritance called a “mixin layer” (Van Hilst and Notkin, 1996; Smaragdakis and Batory, 1998,1999).

One can view the code implementing specific features and aspects as a type of (fine-grain) component, just as templates can be viewed as reusable components or elements, if the "template invocation" is viewed as an interface, just as method and classes can be viewed as simple components. RSEB and FeatuRSEB focus on large-grain, customizable components and frameworks. These components are customized by attaching variants at "variation points." Highly customized, fine-grained aspect-oriented development is complementary to this large-grain component-based development, in which customizable components, communicating through COM/CORBA/RMI interfaces are tied together by a workflow. But well integrated systems cannot be built from arbitrary COTS components. Many of their components must be built or tailored, using fine-grained approaches using inheritance, templates, generators or aspect languages as described above. For further discussion on the relationship between frameworks, customizable components and product-lines, see (Bosch, 1999).

3. A Systematic approach to Feature-Driven Product-Line Development

The systematic approach to feature-driven, component-based product-line development becomes quite simple in concept:

1. Use a feature-driven analysis and design method to develop a feature model, and high-level design with explicit variability and traceability. The feature model is developed in parallel with other models. It can be used to structure the product-line family use cases, and class models. Features are traced to variability in the design and implementation models (Griss, 1998). The design and implementation models will show these explicit patterns of variability.

2. Select one or more aspect-oriented implementation techniques, depending on the granularity of the variable design and implementation features, and the patterns of their combination needed
3. Express the needed aspects as appropriate code fragments that will be combined into complete components using the chosen mechanism – for example as C++ templates, parameters or frames.
4. Design complete applications by selecting and composing features, and then select and compose the corresponding aspects, composing the resulting components and complete application out of code fragments corresponding to these aspects.

4. Examples

At Hewlett-Packard Laboratories, these product-line techniques are being explored in the context of several application prototypes. As described in more detail in (Griss, 2000) several large-scale domain-specific component frameworks for families of medical and banking applications were prototyped using CORBA (and COM) as the distribution substrate, and a mixture of Java and C++ for components. The same experimental component framework architecture was used for two different domains: a distributed medical system supporting doctors in interaction with patient appointments and records, and an online commercial banking system for foreign exchange trades.

The applications in each domain are comprised of several large distributed components, running in a heterogeneous Unix and NT environment. Different components handle such activities as doctor or trader authentication, interruptible session management, task list and workflow management, and data object wrappers. The overall structure of all the components is quite similar both across the domains and across roles within the domains. For example, all of the data object wrapper components are quite similar, differing only in the SQL commands they issue to access the desired data from multiple databases, and in the class-specific constructors they use to consolidate information into a single object. Each component has multiple interfaces. Some interfaces support basic component lifecycle, while others support the specific function of the component (such as session key). Several of the interfaces work together across components to support crosscutting aspects such as transactions, session key management, and data consistency.

Each component is built from one of several skeletal components to provide a shell with standard interfaces and supporting classes. Additional interfaces and classes are added for some of the domain- and role-specific behavior, and then the component is finished by hand. The customization of the components corresponding to the cross-cutting aspects is carried out compatibly across the separate components. Component skeletons are generated by customizing and assembling different parts from a shared parts library. A simplified feature-oriented domain analysis was used to identify the requisite parts and their relationships. This was used to structure a Bassett-style frame-based component generator(Bassett, 1996) which customizes and assembles the parts and components from C++ fragments. Each frame is a template that includes C++ fragments, and directives to select and customize lower-level frames and generate additional fragments of code. Frames are organized to represent the code corresponding to different aspects, such as security, workflow/business process interfaces, transactions, sessions and component lifecycle. Each component interface is represented as a set of C++ classes and interfaces. Component interactions are represented as frames which are then used to generate and compose the corresponding code fragments.

The other major class of prototype applications is agent-based systems for e-commerce and applications management, described in chapter 36 and (Griss, 2000a). We have recognized the need to develop a large variety of different, yet compatible agents. Changing the details of one feature (such as agent communication language, mobility, or conversation management strategy) will require compatible changes across many agents.

Conclusion

In this chapter, I provided an overview of a systematic approach to component-based product-line development. Starting from the set of common and variable features needed to support a product-line, we can systematically develop and assemble the reusable elements needed to produce the customized components and frameworks needed to implement the members of the product-line.

Obviously, the simple prescription presented for product-line CBSE is not a complete method, but does suggests that product-line architects, domain engineers, as well as component and framework designers should now become aware of how to use these ideas to structure their models and designs. I suggest that they then can structure the implementation in a disciplined way to explicitly manage and compose aspects traceable from features into members of the product family.

References

G Arango, “Domain Analysis Methods,” in W. Schäfer et al., *Software Reusability,* Ellis Horwood, Hemel Hempstead, UK, 1994.

PG Bassett *Framing Reuse: Lessons from the Real World,* Prentice Hall 1996.

D Batory and S O'Malley, “The Design and Implementation of Hierarchical Software Systems with Reusable Components”, *ACM Transactions on Software Engineering and Methodology*, 1(4), October 1992, pp. 355-398

D Batory, "Subjectivity and GenVoca Generators," *Proc. ICSR 96*, Orlando, FLA, IEEE, April 1996, pp.166-175.

G Booch, J Rumbaugh and I Jacobson, *The Unified Modeling Language: User Guide*, Addison-Wesley-Longman, 1999.

J Bosch, "Product-Line Architectures and Industry: A Case Study," *Proceedings of International Conference on Software Engineering,* May 1999, Los Angeles, California, USA, ACM press, pp. 544-554.

F Buschmann et. al., *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley and Sons, 1996.

E Gamma, R Helm, R. Johnson and J. Vlissides*, Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

D Garlan and M Shaw, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

ML Griss, "Domain Engineering And Variability In The Reuse-Driven Software Engineering Business," *Object Magazine*, Dec 1996, pp. 67-70.

ML Griss, "Implementing Product-Line Features with Component Reuse", *Proceedings of 6th International Conference on Software Reuse*, Springer-Verlag, Vienna, Austria, June 2000, pp. 137-152.

ML Griss, "How to Implement Product-Line Features By Composing Component Aspects," *Proceedings of 1st Software Product-line Conference*, Denver, Colorado, August 2000a, Kluwer Academic Publishers

ML Griss, J Favaro, and M d'Alessandro, Integrating Feature Modeling with the RSEB. *Proceedings of 5th International Conference on Software Reuse*, Victoria, Canada, June 1998, IEEE, pp. 76-85.

W Harrison and H Ossher, "Subject-Oriented Programming (a critique of pure objects)," *Proceedings of the Object-Oriented Programming, Languages, Systems and Applications conference (OOPSLA 93)*,Washington, DC, ACM, Sep 1993, pp. 411-428.

I Jacobson, ML Griss, and P Jonsson*, Software Reuse: Architecture, Process and Organization for Business Success*, Addison-Wesley-Longman, 1997.

KC Kang et al, “Feature-Oriented Domain Analysis Feasibility Study,” SEI Technical Report CMU/SEI-90-TR-21, November 1990.

KC Kang et. al., “FORM: A feature-oriented reuse method with domain-specific architectures," *Annals of Software Engineering*, V5, Balzer Science Publishers, 1998, pp. 143-168.

KC Kang, S Kim, J Lee and K Lee "Feature-oriented Engineering of PBX Software for Adaptability and Reuseability," *Software - Practice and Experience,* Vol. 29, No. 10, 1999, pp. 875-896.

G Kiczales, J Lamping, A Mendhekar, C Maeda, L Loitinger and J Irwin, "Aspect Oriented Programming," *Proc. ECOOP 1997*, Springer-Verlag, June 1997, pp. 220-242.

P. Kruchten, The 4+1 View Model of Architecture, *IEEE Software*, pp. 42-50, November 1995.

D Musser and A Saini, *STL Tutorial and Reference Guide*, Addison-Wesley, 1996.

DL Parnas, "On the Design and Development Of Program Families*," IEEE Transactions on Software Engineering*, v 2(16), Mar 1976, pp. 1-9.

DL Parnas, "Designing Software for Ease of Extension and Contraction*," IEEE Transactions on Software Engineering*, v 5 (6), Mar 1979, pp. 310-320.

Y Smaragdakis and D Batory, "Implementing Reusable Object-Oriented Components," *Proceedings of 5th International* Conference on Software Reuse, Victoria, BC, June 1998, pp. 36-45.

Y Smaragdakis and D Batory, "Implementing Layered Designs with Mixin Layers," *Proceedings of ECOOP98*, Brussels, Belgium, Springer, July1998a, pp. 1445-1453.

P Tarr, H Ossher, W Harrison, and SM Sutton, Jr., "N degrees of Separation: Multi-dimensional separation of concerns," *Proceedings ICSE 99,* IEEE, Los Angeles, May 1999, ACM press, pp. 107-119.

M Van Hilst and D Notkin, "Decoupling Change From Design," *ACM SIGSOFT*, 1996. Pp. 58-69.

M Van Hilst and D. Notkin, "Using C++ Templates to Implement Role-Based Designs," Japan Society for Software Science and Technology (JSSST) *Symposium on Object Technologies for Advanced Software*, Springer-Verlag, 1996, 1996a, pp. 22-37.

Terminology Sidebar

- A *product-line* is a set of products that share a common set of requirements, but also exhibit significant variability in requirements.
- A *feature* is a product characteristic that users and customers view as important in describing and distinguishing members of the product-line.
- A *framework* is a skeletal product, comprising an implementation of key components, infrastructure and mechanisms that are common to all members of the product-line.
- A *crosscutting* concern or feature results in code that appears in (cuts across) many components. Sometimes, a different decomposition into components can reduce tehamount of crosscutting.
- An *aspect* is a fragment of code, typically a statement, method or template, perhaps parameterized, that will be composed or woven with base code and other aspects into a complete method, class or component to create complete components and products.
- A *variation point* is an RSEB term to describe an explicitly designated location within a component or a model at which a variability mechanism may be used to create a customized component. For example, parameters are used at specified places in the code.