Implementing Product-Line Features with Component Reuse

Martin L. Griss
Software Technology Laboratory
Hewlett-Packard Company, Laboratories
Palo Alto, CA, USA
+1 650 857 8715
martin_griss@hp.com

(Submitted to 6th International Conference on Software Reuse, June 2000)

ABSTRACT
In this paper, we show how the maturation of several technologies for product-line analysis and component design, implementation and customization provides an interesting basis for systematic product-line development. Independent work in the largely separate reuse ("domain analysis") and object-oriented ("code and design") communities has reached the point where integration and rationalization of the activities could yield a coherent approach. We outline a proposed practical path from the set of common and variable features supporting a product-line, to the reusable elements to be combined into customized feature-oriented components and frameworks to implement the products.

Keywords
Product-lines, components, features, frameworks, reuse, domain engineering, aspects, FeatuRSEB

1 INTRODUCTION

Product-lines offer substantial opportunities for reuse

Today, it is increasingly important to manage related products as members of a product-line. Systematic component reuse can play a significant role in reducing costs, decreasing schedule and ensuring commonality of features across the product-line. Business managers can make strategic investments in creating and evolving components that benefit the whole product-line, not just a single product. Studies of effective reuse programs show that most often, successful reuse is associated with a product-line. 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.

A product-line is a set of products that share a common set of requirements, but also exhibit significant variability in requirements. This commonality and variability can be exploited by treating the set of products as a family and decomposing into shared components that separate concerns [1,2,3].

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 amongst optional or alternative requirements, or related to certain product aspects, such as functionality, usability, and performance, or implementation aspects, such as size, execution platform or compatibility with certain standards. Members of a product-line might all exist at one time, or be related to the evolution of 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 Windows or Unix, be large or small, support a small number of files or large number files, etc. Some word processors might have a minimal spell checker, while others could have a large spell checker integrated with a grammar corrector.

A product-line can be built around the set of reusable components. The approach is based on analyzing the products to determine common and variable features, and then developing a product structure and implementation strategy that expresses this commonality in terms of a set of reusable components.

For example, [4] shows how a particular large-scale product-line for a communications product is built not by composing black box components, but by assembling customized subsystems, each based on a "framework" which evolves slowly. Each framework is characterized by a cluster of features, and the subsystem is implemented by modifying parts of the framework and added code

corresponding to the variable features.

Each component will capture some of the set of features, or allow features to be built up by combining components. For example, each word processor in the product-line above might share a common editing component, and optionally one or more components for spell-checking, diagramming, etc. These components may be combined to produce a variety of similar word processors, only some of which would be viable products. Often, these components are used directly without change; in other cases the components are adapted in some way to account for differences in the products that are not expressible by just selecting alternative components; and in yet other cases, the components are substantially modified before inclusion into the product. Thus, some components would be used as is, some would-be parameterized and others would come in sets of plug compatible alternatives with different characteristics to satisfy the features.

Domain analysis helps find features

Domain analysis or domain engineering techniques [5] are used to systematically extract features from existing or planned members of a product-line. Features trees are used to relate features to each other in various ways, showing sub-features, alternative features, optional features, dependent features or conflicting features. These methods, such as FODA [6], FORM [7,8,9], FeatuRSEB [10], FAST, ODM, and Pulse are 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.

Kang [7,8,9] describes the FORM (feature oriented reuse method) as an extension to FODA [6]. FORM leverages FODA into the design and implementation space. FORM includes parameterization of reusable artifacts using features, and a 4-layered feature model: capability, operating and prominent, domain technology, and implementation technique. FORM separates components from component "glue", and synthesis of domain specific components based on feature selection. However, no systematic technique for composing the code from facets is provided.

Once designed, each component is assigned to a component development team which develops and maintains it. Products are assigned to feature teams who select a set of features that define the products, and then select and adapt the appropriate components to cover the features and produce the desired product.

This approach in general is a good plan. It works extremely well when the set of features decomposes nicely into a set of almost independent, fixed components, where very few features span multiple components. For example, products dominated by mathematical and statistical components.

The key problem -Tracing features to components gets complex

Things are not always this simple.

Things get a little more complex when components have compile time or run time parameters that essentially generate a customized component. Here one has to keep track of which parameter settings relate to which version, and to which set of selected features.

Things get much more complicated when the code that implements a particular feature or closely related features needs to be spread across multiple products, classes or modules and is intertwined or tangled with code implementing other (groups of) features. In many (ideal) cases, a particular feature will be implemented as in code that is mostly localized to a single module; but in many other cases features cut across multiple components. Such "crosscutting" concerns result in great difficulty in associating separate concerns with separate components that hide the details.

Example crosscutting features include end-to-end performance, transaction or security guarantees, or end to end functional coherence. In the RSEB[11] and FeatuRSEB approaches[10] to product-line development, we look at or select usecases separately and choose among variant usecase features. In most cases, each usecase typically translates to a collaboration of multiple components, resulting in crosscutting impact. Variability in a single usecase then translates into crosscutting variability in design and implementation components.

In the case of significant crosscutting variability, maintenance and evolution becomes very complicated, since the change in a particular (usecase) feature may require changes all over the resulting software design and implementation .

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.

How should we address this problem? The conventional approach is to build fairly complex maps (essentially hyper-webs) from features to products, and manage the evolution of the features by jointly managing the evolution of the individual components (e.g., see [12].) In the past, this has been done manually, but more recently powerful requirements tracing and management tools have made this somewhat easier.

Even with such traceability tools, what makes this complex is that when one feature changes, code in many modules will have to change; to find and decide how to change this code requires the code corresponding to this feature to be

disentangled from the code for other features. A real pile of spaghetti!

One observation is that typically individual features (or small sets of related features) change more or less independently from other features, and thus a group of code changes can be related to threads drawn from a few related features. This is the same observation that suggest that usecase-driven development is so powerful; it is complete usecases that change or are selected almost independently. Thus separation of concerns is most effective at the higher-level, before "mangling" and "crosscutting" into design or implementation.

2 UNTANGLING THE FEATURE WEB

There are three different approaches that can be used to address this problem. Typically, some combination of these is used in practice. Each approach has benefits and drawbacks.

1. ***Traceability*** - using hyperweb-like tools to help find and manage threads corresponding to one feature (e.g., program slicing, browsers, code coloring. See for example the DOORS or Calibre RM requirements management tools. See also UML language and tool-based traceability [13] and the discussion by Tarr [12] on a traceability hyper-web. In some cases, the threads from requirements through features to design and code are maintained as the software is built; in others, the threads are discovered using maintenance and analysis tools. This is often hard to do, and leads to a complex set of multiple versions of multiple components. When a feature is changed, lots of code in many components is touched, and so many components have to be retested and revalidated after a change.

2. ***Architecture*** - use tools and techniques such as patterns[14,15], styles [16], and technologies such software buses and middleware to provide a good decomposition into separate pieces that are more orthogonal, localizing changes. This often leads to a better structure and a more maintainable, flexible system. However, discovering and maintaining a good design, requires a lot of skill (and some luck) in choosing a decomposition that anticipates changes. Different decompositions are more or less robust against different classes of change. Frequent change often requires significant framework refactoring, moving the implementation of key design aspects to different classes or modules.

3. ***Composition or "weaving" of program fragments*** - using language extensions or a preprocessor, the feature decomposition is used directly to produce pieces of software ("aspects") that are assembled in some manual or mechanical way into complete components or products. Up to now, a systematic approach to this has been lacking, and mixture of manual techniques, with macros or compile time switches has been used to do this in a somewhat clumsy way.

3 FEATURE-DRIVEN, ASPECT-BASED PRODUCT-LINE ENGINEERING.

A systematic approach is emerging. Over the last several years, a number of techniques have been developed to make the "composition and weaving" approach to feature-driven design and development of software practical. The idea is to directly implement some feature (sometimes called aspect when referring to solution or design features) as a fragment of code, and then use some mechanism or tool to combine these fragments (sometimes called facets) together into complete modules.

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 a of the software.

The idea is not to directly manage the resulting components, but to directly manage only the aspects and their related facets, and to dynamically generate a 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 legal ways in which they can be combined. For example, does code get woven 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).

The earlier techniques used macros, classes, inheritance and templates in an ad hoc manner:

- Macros - typically look like procedure calls, but in fact create inline code, overcoming a concern about performance with breaking a system into many small procedures. Some macros can generate alternative pieces of code using conditional expansion, and testing compile time parameters.

- OO inheritance, templates and frameworks. There are several distinct models (or paradigms) for using inheritance and C++ templates - parameterized data types or generics (e.g. STL), parameterized structure inheritance (e.g., frameworks), "mixin layers", 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 supperclasses.

- C++ templates - the body of a method, or of several methods, is created by composing fragments. This was raised to a high-art form in the C++ Standard Template Library (STL), in which a class was constructed from a

composition of an Algorithm, a Container, an Element, and several Adapter aspects.[17]. STL is noteworthy for its lack of inheritance.

More recent work takes a more systematic approach to identifying and representing crosscutting concerns:

- By using multiple-inheritance and layering of "mixin" classes, the subclasses are essentially a method level combination of methods (as class fragments) drawn from several different sources. The subclasses and mixins, can be thought of as the component fragments being woven into complete components or complete applications. It is worth noting that many design patterns also essentially help encapsulate some crosscutting features[14,15])

- Frame-based generators - Paul Bassett [18] shows how a simple frame-based "generator" (he calls it an "assembler") composes code fragments from several frames. A good decomposition results in each frame deals with one or more aspects of the system being generated. Frame technology has been used with impressive results by Netron and clients to implement several product lines rapidly and efficiently.

- System generators - Batory's GenVoca and P++ [19,20]- a specialized preprocessor 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.

- Subject-oriented programming (SOP) - Ossher, Harrison and colleagues [21, 12, 22] represent each concern or feature as a separate package, implemented or designed separately. SOP realizes a model of object oriented code artifacts whose classes, methods and variables are then independently selected, matched and finally combined together non-invasively to produce the complete program." Systems build up as a composition of subjects -- hyper slices -- each a class hierarchy modeling its domain from a particular point of view. Composition rules are specified textually in C++ as templates. An examination of how to extend SOP ideas into design using UML is underway[23].

- Aspect-oriented programming (AOP) - developed by Kiczales and colleagues[24], AOP expands on SOP concepts by providing modular support for programming of crosscutting concerns. Early work focused on domain-specific examples, illustrating how several, nonfunctional, crosscutting concerns, such as concurrency, synchronization, security and distribution properties could be localized. AOP starts with a base component that cleanly encapsulates some application function in code, using methods and classes. AOP then applies one or more (orthogonal) aspects to components to perform large-scale refinements, primarily as design features that modify or crosscut multiple base components. Aspects are implemented using an aspect language which makes insertions and modifications at defined join points (similar to UML extension points or RSEB variation points), which may be as generic as constructs in the host programming language or as specific as event patterns or code markers unique to a particular application. One such language, AspectJ, extends Java with statements such as "*crosscut*" to identify places in Java source or event patterns. The statement "*advise*" then inserts new code or modifies existing code whereever it occurs in the program. AspectJ thus weaves aspect extensions into the base code, modifying and extending a relatively complete program.

These added and combined code fragments are sometimes called facets or are referred to by their corresponding aspects, subjects or concerns expressing the features, depending on the granularity and how they will be woven together. As experience with C++ templates has grown, it has been shown that some of the weaving originally thought to need a system generator or special aspect-oriented generator, could in fact be accomplished by clever decomposition into template elements, using a combination of C++ templates and inheritance called a "mixin layer". For example, work by VanHilst and Notkin [25, 26] using templates to implement role mixins, and later work by Smaragdakis and Batory[27,28,29] showing how P++ and Genvoca style composition [19,20] can use a more powerful C++ template-based mixin layer idiom, and so avoid the need for a special generator (though more disciplined use seems needed).

Smaragdakis and Batory[27,28,29] show how to implement a collaboration between several OO-components as a programming pattern known as a mixin layer, using several class templates that the net result in layer is of super classes that can be easily composed. This approach simplifies and generalize this the work by Van Hilst and Notkin [26,30] . In this approach an application is expressed as a composition of independently-definable collaborations. In this way each object or application play is a collaboration of several roles. As in UML and RSEB usecase-driven design, each role gives rise to separate code that must then be woven into the complete system. The mixin-layer approach makes the contribution of each role (feature) distinct and visible across multiple objects.

Cardone [31] shows how both aspect oriented programming and Genvoca begin with some encapsulated representation of system function (AOP component, GenVoca realm) and then provides a mechanism to define and apply large-scale refinements (AOP aspects, GenVoca components) at specific sites (AOP join points, parameters in GenVoca components). The AOP weaver and the GenVoca generator perform transformations from the aspect languages or type expressions. Both are implemented as pre-processors.

Related work has been done by Lieberherr [32] on Demeter's flexible composition of program fragments, and by Aksit[32] and others.

Tarr [12] shows how separation of concerns can lead to a hyperweb tracing from specific concerns through parts of multiple components. Each slice (hyperslice) through this web corresponds to a different composition/decomposition of the application. She suggests that there are a variety of different ways to decompose the system, and this stresses that multidimensional separation. This is the "tyranny of the dominant decomposition". - no single means of decomposition, including features, is always adequate, as developers may have other concerns instead/in addition to features (or functions, or classes, or whatever). Neither AOP nor SOP provides true support for multiple dimensions; AOP supports essentially 1.5 dimensions, and SOP supports multiple dimensions, but only a fixed number at a time. In both cases, once you've chosen the dominant dimension(s), you're stuck with them.

Tarr suggests that these multiple decompositions can be best represented as hyper-slices in a hyper-web across the aspects of components. Each aspect is a hyper slice and a set of aspects together with core classes approximate a hyper model. This is essentially equivalent to the traceability expressed in UML [13], RSEB[11] and FeatuRSEB [10]. It allows tracing from features to crosscutting aspects within multiple components.

One can view the code implementing specific features (facets) a type of (fine-grain) component, just as templates can be viewed as reusable components. In general though, as done in RSEB and FeatuRSEB, we focus on large-grain, customizable components and frameworks (*aka* Component Systems), in which variants are "attached" at "variation points." We take the perspective that highly-customized, fine-grained facet-based development can be seen as complementary to large-grain component-based development. Large systems are made of large grained customizable components (potentially distributed, communicating through COM/CORBA/RMI interfaces and tied together with a workflow glue). But well integrated systems cannot be built from arbitrary [C]OTS components. Many of their components must be built or tailored, using fine grained approaches, such as aspect and facet based development described. Facet based development is not only limited to smaller products, but is used for the customization of the components of large products.

4 FEATURE ENGINEERING DRIVES ASPECT ENGINEERING.

Thus the systematic approach becomes quite simple in concept:

1. Use a feature-driven analysis and design method, such as FeatuRSEB to develop a feature model and high-level design with explicit variability and traceability. The feature diagram is developed in parallel with other diagrams as an index and structure for the product-line family usecases, and class models. Features are traced to variability in the design and implementation models [Griss98]. The design and implementation models will thus show these explicit patterns of variability.

   Recall that a use case maps into a collaboration in UML. FeatuRSEB manages both a use case model (with variation points and variants) and a feature model, and through traceability, relates the roles and variation points in the use case, to feature model variants, and then to design and implementation elements.
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.
3. Express the aspects/facets using the chosen mechanism
4. Design complete applications by selecting and composing features, which then select and compose aspects, weaving the resulting components and complete application out of corresponding fragments.

Obviously, this simple prescription is not a complete method, but does suggests that product-line architects, domain engineers and component and framework designers should now become aware of facet, subject and aspect engineering (and vice-versa) and begin to use these ideas to structure their models and designs. Then they can structure the implementation, using OO inheritance and templates in a disciplined way, or perhaps use one of the generators, to explicitly manage and compose aspects traceable from features into members of the product family.

At Hewlett-Packard Laboratories we have prototyped several large-scale domain specific component frameworks, using CORBA (and COM) as the distribution substrate, and a mixture of Java and C++ for components. Components have multiple interfaces, several of which are designed consistently to support crosscutting feature sets. Each component is manually finished, starting from a customized component skeleton generated using a Basset-style frame generator. The frames are used to represent different facets, 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, with interactions represented as frames which are then used to generate and composed the corresponding facet code.

5 SUMMARY AND CONCLUSIONS

Feature oriented domain analysis and design techniques for product-lines and aspect/facet-oriented implementation technologies have matured to the point that it seems possible to create new, clear and practical path for product-line implementation. That is, 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 to implement the products.

ACKNOWLEDGEMENTS
I am grateful for several useful and extremely clarifying suggestions and comments on early drafts of this paper by Gregor Kiczales, Peri Tarr and Michael Van Hilst.

REFERENCES

[1] DL Parnas, "On the criteria to be used in decomposing systems into modules," CACM, 15(12):1053-1058, Dec 1972.

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

[3] DL Parnas, "Designing Software for Ease of Extension and Contraction," IEEE Transactions on Software Engineering, v 5, #6, pp 310-320, Mar 1979.

[4] J Bosch, "Product-Line Architectures and Industry: A Case Study," Proceedings of ICSE 99, 16-22 May 99, Los Angeles, California, USA, ACM press, pp. 544-554.

[5] G Arango, "Domain Analysis Methods," in W. Schäfer et al., Software Reusability, Ellis Horwood, Hemel Hempstead, UK, 1994.

[6] KC Kang et al, "Feature-Oriented Domain Analysis Feasibility Study," SEI Technical Report CMU/SEI-90-TR-21, November 1990.

[7] KC Kang et al, "FORM: A feature-oriented reuse method with domain-specific architectures," Annals of Software Engineering, V5, pp 143-168, 1998.

[8] KC Kang, "Feature-oriented Development of Applications for a Domain", Proceedings of Fifth International Conference on Software Reuse, June 2 - 5, 1998. Victoria, British Columbia, Canada. IEEE computer society press, pp. 354-355.

[9] 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, pp. 875-896, 1999.

[10] M Griss, J Favaro and M D'allesandro, "Integrating Feature Modeling with the RSEB," Proceedings of ICSR98, Victoria, BC, IEEE, June 1998, pp. 36-45.

[11] I Jacobson, M Griss, and P Jonsson, Software Reuse: Architecture, Process and Organization for Business Success, Addison-Wesley-Longman, 1997.

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

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

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

[15] F Buschmann et. al. "Pattern-Oriented Software Architecture - A System of Patterns." John Wiley & sons, 1996.

[16] D Garlan and M Shaw, Software Architecture, Perspectives on an Emerging Discipline, Prentice-Hall, 1996.

[17] D Musser and A Saini, STL Tutorial and Reference Guide, Addison-Wesley, 1996.

[18] PG Bassett Framing Reuse: Lessons from the Real World, Prentice Hall 1996.

[19] 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):355-398, October 1992.

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

[21] W Harrison and H Ossher, "Subject-Oriented Programming (a critique of pure objects)," Proc. OOPSLA 93Washington, DC, Sep 1993, 411-428, ACM.

[22] RJ Walker, ELA Banniassad and GC Murphy, "An Initial Assessment of Aspect Oriented Programming," Proc. ICSE 99, IEEE, Las Angeles, May 1999, 120-130.

[23] S Clarke, "Subject Oriented Design," Submitted to OOPSLA 1999.

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

[25] M. Van Hilst and D. Notkin, "Using C++ Templates to Implement Role-Based Designs," JSSST Symposium on Object technologies for Advanced Software, Springer-Verlag, 1996, 22-37

[26] M Van Hilst and D Notkin, "Using Role Components to Implement Collaboration-Based Designs," Proc. OOPSLA96, 1996, 359-369.

[27] Y Smaragdakis and D Batory, "Implementing Reusable Object-Oriented Components," Proc. of ICSR 98, Victoria, BC, June 1998, pp36-45.

[28] Y Smaragdakis and D Batory, "Implementing Layered Designs with Mixin Layers," Proc. of ECOOP98, 1998.

[29] Y Smaragdakis, "Reusable Object-Oriented

Components," Proceedings of Ninth Annual Workshop on Software Reuse, Jan. 7-9, 1999, Austin, Texas.

[30] M Van Hilst and D Notkin, "Decoupling Change From Design," ACM SIGSOFT, 1996.

[31] R Cardone, "On the Relationship of Aspect Oriented Programming and GenVoca," Proc. WISR, Austin Texas, 1999

[32] KJ Lieberherr and C Xiao, "Object-oriented Software Evolution," IEEE Transactions on Software Engineering, v 19, No. 4, pp. 313-343, April 1993.

[33] M Aksit, L Bergmans, and S Vural, "An Object Oriented Language-Database Integration Model: the Composition-Filters Approach," in Proceedings of the 1992 ECOOP, pp. 372-395, 1992.