

Implementing Product-Line Features with Component Reuse

Martin L. Griss

Hewlett-Packard Company, Laboratories
Palo Alto, CA, USA
+1 650 857 8715
martin_griss@hp.com

To appear, Proc. 6th International Conference on
Software Reuse, Vienna, Austria, Jun 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 OO ("code and design") communities has reached the point where integration and rationalization of the activities could yield a coherent approach. We outline a proposed 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.

1 Introduction

Today, it is increasingly important to manage related products as members of a product-line. Rapid development, agility and differentiation are key to meeting increased customer demands. 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. 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 through shared components.

Many different authors have worked on this area from multiple perspectives, resulting in somewhat confusing and overlapping terminology. We will define some terms as we go, and collect other terms at the end of the paper.

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 the

design and implementation of the products into a set of 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 characteristics, such as functionality, usability, and performance, or implementation characteristics, 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.

A product-line can be built around a set of reusable components. The approach is based on analyzing the products to determine common and variable features, and then developing a product structure, software architecture and implementation strategy that expresses this commonality in terms of a set of reusable components. The decision to do so involves a combination of economic and technical issues - there must be a sufficient number of different members of the product-line, closely enough related, to justify the extra work of developing appropriately customizable reusable components.

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. As another example, consider all of the many kinds of printers sold by HP. There are black and white printers, and color printers. Some use laser technology, others use inkjet technology. Some are small, while others are large, with multiple paper handlers and so on. Marketing decides which families to treat as a product-line, while engineering decides which should share a common firmware architecture and components.

For a final 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 adding code corresponding to the variable features.

Each component will capture a subset of the features, or allow individual 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.

1.1 Domain Analysis Helps Find Features

Domain analysis and 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], FeatuRSEB [7], FORM [8,9,10], 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.

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 computation are easier to implement from relatively fixed components, since the interactions between features, and hence components, is less complex and better understood than in some other domains.

1.2 Tracing Features to Components is 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 several closely related features needs to be spread across multiple products, subsystems, modules or classes and is intertwined or tangled with code implementing other (groups of) features. In many (ideal) cases, a particular feature will be implemented as code that is mostly localized to a single module; but in many other cases features cut across multiple components. Such "crosscutting" concerns make it very difficult to associate separate concerns with separate components that hide the details.

Examples of crosscutting features include end-to-end performance, transaction or security guarantees, or end to end functional coherence. To produce a member of the product line that meets a particular specification of such a cross cutting feature, requires that several (or even all) subsystems and components make compatible choices, and include specifically related fragments of code.

The RSEB[11] and FeatuRSEB approaches[7] to product-line development are “usecase-driven.” This means that we first structure the requirements as a set of distinct usecases that are largely independent. Design and implementation of each usecase typically gives rise to a collaboration of multiple classes, resulting in the specification of a set of compatible responsibilities for these classes. Some classes participate in multiple collaborations, requiring the merging of the cross-cutting contributions to each class. For product-line development, commonality and variability analysis in RSEB and FeatuRSEB structures each usecase in terms of core and variant parts of the usecase. FeatuRSEB augments the RSEB usecases with an explicit FODA-derived feature model to highlight commonality and variability of features. The variability structure in the usecase (and feature) models then give rise to variant parts of the corresponding classes, to be attached at designated “variation points.” Thus selection of a single variant in a usecase, gives rise to multiple variant selections in the resulting classes. These choices need to be made compatibly to match the high-level crosscutting concern.

Several classes and some variants are packaged into a single component, and thus variability in a single usecase translates into crosscutting variability in multiple design and implementation components.

Kang [8,9,10] describes the FORM (feature oriented reuse method) as an extension of FODA [6] to better support feature-driven development. FORM extends FODA into the design and implementation space. FORM includes parameterization of reusable artifacts using features. The FORM feature model is structured in four layers, to better separate application-specific features from domain, operating environment or implementation features. These layers reflect a staged transition from product-line features through key domains to detailed implementation elements. FORM encourages synthesis of domain specific components from selected features. However, no systematic technique for composing the code from related aspects is provided.

In the case of significant crosscutting variability, maintenance and evolution become 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 [31].) 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 task so 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.

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 led to usecase-driven development; 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, such as the DOORS or Calibre RM requirements management tools.) See also UML language and tool-based traceability [12] and the discussion by Tarr [31] 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, code in many components is changed, and many components have to be retested and revalidated.
2. **Architecture** - use tools and techniques such as patterns[13,14], styles [18], and flexibility-enhancing 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 an appropriate decomposition that anticipates the kinds of changes that will arise during the evolution of a system. Different decompositions are more or less robust against different classes of change. For example, consider the well-known differences in flexibility of the functional decomposition or the object-oriented decomposition of a system, in terms of how they encapsulate or hide decisions. In either case, adding new behavior or new information fields might result in only a local change, a more radical global change that might affect only multiple methods in one class, or a single change in one data structure that might affect many procedures all over the system. This kind of frequent non-local change often requires significant framework refactoring to change the dominant decomposition so as to (re-)group together or further factor those parts of the design that seem be dependent, moving the implementation of key design decisions 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 components, subsystems or complete products. The traditional approach, using a combination of manual techniques to select and combine features, supported with mechanisms based on macros or compile time parameters is somewhat clumsy and error-prone.

3 Feature-Driven, Aspect-Based Product-Line Engineering.

A systematic, automated approach is emerging. Over the last several years, several 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 also called *aspect*) as several fragments of code, and then use some mechanism or tool to combine these code fragments together into complete modules.

The key idea is to identify a useful class of program fragments that can be mechanically combined using some tool, and associating these fragments with distinct features of the software. One does not directly manage the resulting components, but instead directly manages only the common and variable features, the design aspects and their corresponding implementation code fragments. One then dynamically generates a custom 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 or woven together. For example, code could be woven at the module or class level, at the method level or procedure level, or even within individual statements. The approach could be expressed only as a "method" (a disciplined way of using standard language features such as templates or inheritance) or can be expressed and enforced by a new tool that effectively changes the language (such as a preprocessor, language extension or macro).

The following brief summary of these techniques is not meant to give a complete history of the many threads of work in the field, but just to highlight some of the developments. This paper can only provide a brief summary of several "intuitively" related techniques. While this is not a coherent, comprehensive analysis or integration, we hope this will inspire others to push the goal of integration further.

Starting from Parnas' papers on separation of concerns and hiding[1,2,3] there have been numerous efforts to design methods, languages, macros, tools and environments to make the specification and implementation of layered abstractions easier. Languages such as Modula, Algol-68, and EL/1, Goguen's work on parameterized programming[16], tools such as the programmer's apprentice, and many program transformation systems are notable. Meta-programming was first popularized in the LISP community in the form of LISP computational macros, followed by the work of Kiczales and colleagues on the meta-object protocol in CLOS and other languages[15,17]. "Mixin" classes, and "before" and "after" methods were also used in LISP through stylized multiple inheritance.

Earlier techniques used procedures, macros, classes, inheritance and templates in an *ad hoc* manner. Macros look like procedure calls, but 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.

There are several distinct models (or paradigms) for using OO inheritance and C++ templates - parameterized data types or generics (e.g. STL), structure inheritance (e.g., frameworks), and parameterized algorithms and/or compile time computation (e.g. for tailoring matrix algorithms). In the case of (C++) frameworks, subclasses define new methods that "override" the definition of some default methods inherited from super-classes.

More recent work takes a more systematic approach to identifying and representing crosscutting concerns and component (de-)composition. Stepanov and Musser converted their earlier Ada Generic Programming work to C++[19,20], at a time when C++ template technology was weaker and still largely untried. Batory and his students developed the GenVoca system generator and the P++ preprocessor language for component composition[21].

Subsequent work on Generic Programming (GP) by Stepanov[19,20,22], Jazayeri[23], C++ template role mixins by Van Hilst and Notkin[24,25,26] and C++ template meta-programming by Czarnecki and Eisenecker[27,28,29], and others showed the power of these techniques. Recently, GP has started to change from meaning Generic Programming to mean Generative Programming, including domain analysis[29].

In discussing several of these techniques in more detail, the intent is not to advocate or denigrate any approach; rather, the goal is to show how they are similar attempts to explicitly express the code corresponding to features as separate, composable software fragments. Each approaches the many issues from different starting points and perspectives. Each technique is also changing over time, as experience grows, and the groups influence each other. When components entered the picture, several different technologies began to come together.

- C++ templates and generic programming - the body of a method, or of several methods, is created by composing fragments. This was heavily exploited 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 elements[19,20]. STL is noteworthy for its lack of inheritance.
- C++ template role and layer "mixins" - By using multiple-inheritance or layering of "mixin" classes, the subclasses are essentially a method level combination of methods (as class fragments) drawn from several different sources. The base classes and mixins can be thought of as 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[13,14,31].
- Frame-based generators - Paul Bassett [30] shows how a simple frame-based "generator" (he calls it an "assembler") composes code fragments from several frames, linked into a form of "inheritance" or "delegation" hierarchy. A frame is a

textual template that includes a mixture of code fragments, parameterized code, and generator directives to select and customize lower-level frames, and to incorporate additional code fragments generated by the processing of lower frames. While quite ad hoc frame structures are possible to achieve quite complex code transformation and generation, a good decomposition results in each frame dealing with only one or a small number of distinct aspects of the system being generated. Frame technology has been used with impressive results by Netron and its clients to implement several product lines rapidly and efficiently. It has also been used by HP labs in our customizable component framework work, discussed in more detail below.

- Subject-oriented programming (SOP) - Ossher, Harrison and colleagues [31,32,33] 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. A subject is a collection of classes, defining a particular view of a domain or a coherent set of functionality. For example, a subject might be a complete class or class fragment, a pattern, a feature, a component or a complete (sub)system. Composition rules are specified textually in C++ as templates. SOP techniques are being integrated with UML to align and structure requirements, design and code around subjects, representing the overlapping, design subjects as stereotyped packages[45].
- Aspect-oriented programming (AOP) - developed by Kiczales and colleagues[36], AOP expands on SOP concepts by providing modular support for direct programming of crosscutting concerns[44]. 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 (or class) that cleanly encapsulates some application function in code, using methods and classes. AOP then applies one or more aspects (which are largely orthogonal if well designed) to components to perform large-scale refinements which add or change methods, 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. Join points are explicitly defined locations in the base code at which insertions or modifications may occur, similar to UML extension points or RSEB variation points. These join points 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[32], 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 where ever it occurs in the program. AspectJ weaves aspect extensions into the base code, modifying and extending a relatively complete program into a newer program. Thus AspectJ deals with explicit modification or insertion of code into other code, performing a refinement or transformation.

- System generators - Batory's GenVoca and P++ [21,34]- a specialized preprocessor assembles systems from specifications of components, and composition as layers. GenVoca eschews the arbitrary code transformations and refinements possible with AOP and SOP, instead carefully defining (e.g., with a typing expression), composing layered abstractions, and validating compositions. These techniques have been applied to customizable databases, compilers, data structure libraries and other domains.

Related work has been done by Lieberherr and colleagues [34,43] on Demeter's flexible composition of program fragments, and by Aksit[37,38] and others.

3.1 Evolution and Integration of the Techniques

As experience with C++ templates has grown, some of the weaving originally thought to need a system generator or special aspect-oriented language, can in fact be accomplished by clever decomposition into template elements, using a combination of nested C++ templates and inheritance.

VanHilst and Notkin [24,25,26] use templates to implement role mixins. They show how to implement a collaboration between several OO-components using role mixins. Batory and Smaragdakis simplified and extended the role mixin idea into mixin layers[39,40,41]. Using parameterized sets of classes and templates, groups of application classes are built up from layer of super-classes that can be easily composed. They show how P++ and GenVoca style composition can use C++ template mixin layers, and so avoid the need for a special generator. However, a disciplined use of the C++ templates seems needed in order to maintain the layering originally ensured by the specialized P++ language.

As in RSEB and FeatureRSEB, these role and mixin layer techniques express an application as a composition of independently defined collaborations, and a portion of the code in a class is derived from the specific role that the class plays in a collaboration of several roles. The responsibilities associated with each role played by an object gives rise to separate code that must then be woven into the complete system. The mixin-layer approach to implementation makes the contribution of each role (feature) distinct and visible across multiple classes, all the way to detailed implementation, rather than combining the code in an untraceable manner as was done in the past.

Cardone [42] 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 have been implemented as pre-processors (AspectJ and P++), but more recently, GenVoca has moved to a stylized use of C++ templates.

Tarr [31] 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 any particular 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, changing decomposition dimensions; AOP supports essentially one primary decomposition dimension (e.g., class structure of the base program) with modifying aspects, while SOP supports only a fixed number of multiple dimensions, based on the original packaging decomposition. In both cases, once a dominant dimension(s) and corresponding primary decomposition of the software is chosen, all subsequent additions and modifications corresponding to a particular aspect will have to be structured correspondingly. This means that in some cases, what is conceptually a completely orthogonal aspect will result in some code fragments that will be aware of the existence and structure of the other aspects. See also Van Hilst's discussion of the Parnas' KWIK example[26].

Tarr suggests that these multiple decompositions can be best represented as hyper-slices in a hyper-web across the aspects of the 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 [12], RSEB[11] and FeatuRSEB[7]. It allows tracing from features to crosscutting aspects within multiple components.

3.2 Disentangling the Terminology

Many different people have worked on this problem from different perspectives, without being aware of or taking account of each other's work. Hence there is a considerable amount of overlapping terminology, and great care must be taken to be sure the right interpretation of words such as feature, aspect, framework, mixin or role are used for any given paper. It is not the goal of this paper to create a new and completely consistent set of terms, nor to completely describe the subtle nuances between related terms. Unfortunately the paper can not avoid using the terms as used by the original authors. Many of the terms are used to indicate various forms of packaging, management or granularity of different collections or slices of software workproducts created during the development of a software system.

More work needs to be done to clarify the exact relationship between the terms feature, aspect, refinement, and mixin. In some cases, the differences are not very significant, while in others one could either be referring to the design concept or instead to its implementation (like the confusion between the terms "architecture" and "framework").

As far as possible, we use the UML[12]and RSEB[11] definitions.

- *System* - A complete set of software workproducts, mostly self-contained, and able to be understood largely independent of other systems. May be formally described by a set of UML models. Some of the workproducts in a system may be packaged

for sale and deployment as a product, in the form of a set of applications and processes that work together in a distributed context.

- *Workproduct* - Any document used or produced during the production of a software system. These include models, designs, architectures, templates, interfaces, tests, and of course code.
- *Product* - The subset of the system workproducts that are packaged for sale or distribution, and managed as a complete unit by the development organization.
- *Product-line* - A set of products that share a common set of requirements and significant variability. Members of a product-line can be treated as a program family, and developed and managed together to achieve economic, marketing and engineering coherence and efficiency.
- *Subsystem* - A subset of the workproducts produced during the hierarchical decomposition of a system, packaged as a self-contained system in its own right. It is explicitly identified as a part of the larger system, encapsulating key software elements into a unit.
- *Component* - The smallest unit of software packaged for reuse and independent deployment, though typically must be combined with other components and a framework to yield a complete product. Components should be carefully packaged, well documented, well-designed units of software, with defined interfaces, some of which are prescribed for compliance with a specific framework. For example, COM components, CORBA components and JavaBeans are three different standard component models. Each uses a different style of interface specification and has a different set of required interfaces. Other component models can be defined by a particular product-line architecture.
- *Component system* - An RSEB term to reflect that components are most often not used alone, but with other compatible components and thus will be designed, developed and packaged together as a system, with the individual components represented by subsystems in the component system.
- *Framework* - Loosely speaking, a framework is a reusable "skeletal," or "incomplete" system, that is completed by the addition of other software elements. Note that a subroutine or class library also provides reusable software elements, but plays a subservient role to the additional software, while a framework plays a dominant role as a reusable software element that captures and embodies the architecture and key common functionality of all systems built from the framework. Typically, the framework supplies the major control structures of the resulting system, and thus "calls" the additions, while the subroutine or class library provides elements designed to be "called" by the supplied additions. Different kinds of framework use different kinds of additional elements and completion mechanisms. For example, the well-known Johnson and Foote[46] framework provides the top levels of a class structure, providing abstract and concrete base classes. Additions are provided in the form of added subclasses that inherit from the supplied base classes. These added subclasses sometime override default behavior provided by supplied methods. In a well defined framework, key collaborations and relationships are built into the supplied classes, so that the added classes can be simpler and largely independent. Other framework styles depend on different

mechanisms, such as templates and "plug in" interfaces, to allow the additions to be attached, and use the services provided by the framework.

- *Feature* - A product characteristic used to describe or distinguish members of a product line. Some features relate to end-user visible characteristic, while others relate more to system structure and capabilities.
- *Aspect* - In AOP, the set of class additions or modifications to a base class; also, the orthogonal concept giving rise to the set of adding or modifying software elements.
- *Mixin* - Usually refers to a class that is used in a multiple inheritance situation only to augment a base class, and not for use by itself. The term was introduced in the LISP Flavors object system.
- *Refinement* - The application of a transformation step to some software workproduct (a model or code) to produce a more detailed, less generic and less abstract software product
- *Domain* - A coherent problem area. This is often characterized by a domain model, perhaps in the form of a feature diagram, showing common, alternative and optional features. Domain analysis is concerned with identifying the domains(s) present in a problem and its solution(s) and producing a model of the domain, its commonality and variability structure, and its relationship to other domains.

4 Feature Engineering Drives Aspect Engineering.

Thus the systematic approach becomes quite simple in concept:

- a. 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 [7]. The design and implementation models will thus show these explicit patterns of variability. Recall that a usecase maps into a collaboration in UML. FeatuRSEB manages both a usecase model (with variation points and variants) and a feature model, and through traceability, relates the roles and variation points in the usecase, to feature model variants, and then to design and implementation elements.
- b. Select an aspect-oriented implementation technique, depending on the granularity of the variable design and implementation features, the patterns of their combination, and the process/technology maturity of the reusers.
- c. Express the aspects and code fragments using the chosen mechanism
- d. 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.

One can view the code implementing specific features as a type of (fine-grain) component, just as templates can be viewed as reusable components [26]. In general though, 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 aspect-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). It is well known that integrated systems cannot be built from arbitrary [C]OTS components[47]. On the one hand, these components must be built using consistent shared services and mechanisms, and interface style, suggesting a common decomposition and macro-architecture. On the other hand, many of the components must be built from scratch or tailored to meet the specific needs of the application. This needs to be done using compatible fine-grained approaches, since many of the components must be customized compatibly to achieve the constraint of the cross-cutting concern.

This means that the feature-driven aspect-based development should not be limited only to smaller products, but should also be used for the (compatible) customization of the components of large products.

4.1 Hewlett-Packard Laboratories Component Framework

At Hewlett-Packard Laboratories we have prototyped several large-scale domain specific component frameworks, using CORBA and COM as the distribution substrates, and a mixture of Java and C++ for components. The same 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 (for examples as doctors switch from patient to patient, or move from terminal to terminal), 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, 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 or basic 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 some code is written by hand to finish the component. It is important that the customization of the components corresponding to the cross-cutting aspects be carried out compatibly across the separate components. The skeletal component is essentially a small framework for a set of related components. Basic and specific component skeletons are generated by customizing and assembling different parts from a shared parts library. Some parts are specific to one of the cross-cutting aspects, while others are specific to implementation on the target platform. We did a simplified feature-oriented domain analysis to identify the requisite parts and their relationships, and used this to structure a component generator which customizes and assembles the parts and components from C++ fragments. We used a modified Basset frame generator, written in Perl. The input is expressed as set of linked frames, each a textual template that includes C++ fragments and directives to select and customize lower-level frames and generate additional fragments of code. The frames were organized to mostly 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, with interactions represented as frames which are then used to generate and composed the corresponding code fragments.

The components are then manually finished by adding code fragments and complete subclasses at indicated points in the generated code.

5 Summary and Conclusions

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

The simple prescription give in section 4 is not a complete method, but does suggests that product-line architects, domain engineers and component and framework designers should now become aware of 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.

More work also needs to be done on clarifying the terminology, and unifying several of the approaches.

Acknowledgements

I am grateful for several useful and extremely clarifying suggestions and comments on revisions of this paper by Don Batory, 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. M Griss, J Favaro and M d'Alessandro, "Integrating Feature Modeling with the RSEB," *Proceedings of ICSR98*, Victoria, BC, IEEE, June 1998, pp. 36-45.
8. KC Kang et al, "FORM: A feature-oriented reuse method with domain-specific architectures," *Annals of Software Engineering*, V5, pp 143-168, 1998.
9. 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.
10. KC Kang, S Kim, J Lee and K Lee "Feature-oriented Engineering of PBX Software for Adaptability and Reusability", *Software - Practice and Experience*, Vol. 29, No. 10, pp. 875-896, 1999.
11. I Jacobson, M Griss, and P Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*, Addison-Wesley-Longman, 1997.
12. G Booch, J Rumbaugh and I Jacobson, *The Unified Modeling Language: User Guide*, Addison-Wesley-Longman, 1999.
13. E Gamma, R Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
14. F Buschmann et. al. *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley & sons, 1996.
15. G Kiczales, JM Ashley, L Rodriguez, A Vahdat, and DG Bobrow, "Metaobject protocols: Why we want them and what else they can do," in A. Paepcke, editor, *Object-Oriented Programming: The CLOS Perspective*, pages 101--118. The MIT Press, Cambridge, MA, 1993.
16. JA Goguen, "Parameterized Programming," *IEEE Trans. Software Eng.*, SE-10(5), Sep 1984, pp. 528-543."
17. G Kiczales, J des Rivières, and DG Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

18. D Garlan and M Shaw, *Software Architecture, Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.
19. DR Musser and AA Stepanov. Algorithm-Oriented Generic Libraries. In *Software Practice and Experience*, Vol 24(7), 1994
20. DR Musser and A Saini, *STL Tutorial and Reference Guide*, Addison-Wesley, 1996.
21. 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.
22. J Dehnert and A Stepanov, "Fundamentals of Generic Programming," *Proc. Dagstuhl Seminar on Generic Programming*, April 27--May 1, 1998, [Schloß Dagstuhl](#), Wadern, Germany.
23. M Jazayeri, Evaluating Generic Programming in Practice, *Proc. Dagstuhl Seminar on Generic Programming*, April 27--May 1, 1998, [Schloß Dagstuhl](#), Wadern, Germany.
24. 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
25. M Van Hilst and D Notkin, "Using Role Components to Implement Collaboration-Based Designs," *Proc. OOPSLA96*, 1996, 359-369.
26. M Van Hilst and D Notkin, "Decoupling Change From Design," *ACM SIGSOFT*, 1996.
27. K Czarnecki and UW Eisenecker, "Components and Generative Programming, ACM SIGSOFT 1999 (ESEC/FSE), LNCS 1687, Springer-Verlag, 1999. Invited talk.
28. K Czarnecki and UW Eisenecker. Template-Metaprogramming. Available at <http://home.t-online.de/home/Ulrich.Eisenecker/meta.htm>
29. U Eisenecker, "Generative Programming: Beyond Generic Programming", *Proc. Dagstuhl Seminar on Generic Programming*, April 27--May 1, 1998, [Schloß Dagstuhl](#), Wadern, Germany.
30. PG Bassett, *Framing Reuse: Lessons from the Real World*, Prentice Hall 1996.
31. 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.
32. CV Lopes and G Kiczales, Recent Developments in AspectJ™, In *ECOOP'98 Workshop Reader*, Springer-Verlag LNCS 1543.
33. W Harrison and H Ossher, "Subject-Oriented Programming (a critique of pure objects)," *Proc. OOPSLA 93*, Washington, DC, Sep 1993, 411-428, ACM.
34. D Batory, "Subjectivity and GenVoca Generators," *Proc. ICSR 96*, Orlando, FLA, IEEE, April 1996, pp.166-175.
35. KJ Lieberherr and C Xiao, "Object-oriented Software Evolution," *IEEE Transactions on Software Engineering*, v 19, No. 4, pp. 313-343, April 1993.
36. 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.
37. 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.
38. Mehmet Aksit. Composition and Separation of Concerns in the Object-Oriented Model. In *ACM Computing Surveys* 28A(4), December 1996.
39. Y Smaragdakis and D Batory, "Implementing Reusable Object-Oriented Components," *Proc. of ICSR 98*, Victoria, BC, June 1998, pp. 36-45.
40. Y Smaragdakis and D Batory, "Implementing Layered Designs with Mixin Layers," *Proc. of ECOOP98*, 1998.

41. Y Smaragdakis, "Reusable Object-Oriented Components," *Proceedings of Ninth Annual Workshop on Software Reuse*, Jan. 7-9, 1999, Austin, Texas.
42. R Cardone, "On the Relationship of Aspect Oriented Programming and GenVoca," *Proc. WISR*, Austin Texas, 1999.
43. M Mezini and K Lieberherr, "Adaptive Plug-and-Play Components for Evolutionary Software Development," *Proceedings of OOPSLA'98*, pp.97-116
44. RJ Walker, ELA Banniassad and GC Murphy, "An Initial Assessment of Aspect Oriented Programming," *Proc. ICSE 99*, IEEE, Las Angeles, May 1999, pp. 120-130.
45. S Clarke, W Harrison, H Ossher and P Tarr "Towards Improved Alignment of Requirements, Design and Code", in *Proceedings of OOPSLA 1999*, ACM, 1999, pp. 325-339.
46. R Johnson and B Foote, "Designing reusable classes." *Journal of Object-Oriented Programming*, pp. 22-30, 35, June 1988.
47. D Garlan, R Allen, and John Ockerbloom, "Architectural mismatch, or Why it's hard to build systems out of existing parts." In *Proc. ICSE*, 1995.