Implementing Product-Line Features By Composing Component Aspects

Martin L. Griss
Laboratory Scientist, Software Technology Laboratory, Hewlett-Packard Company, Laboratories, Palo Alto, CA., 94304-1126, USA, 650-857-8715, griss@hpl.hp.com

(To appear, Proceedings of First International Software Product Line Conference, Denver, CO, Aug 2000.)

Key words: Product-lines, components, features, frameworks, reuse, domain engineering, aspects, FeatuRSEB, agents, e-commerce, generators.

Abstract: Emerging E-Commerce systems are highly dynamic and require even more flexibility and reduced time to market than that traditionally provided by product-line and component-based development (CBSE). In this paper we describe our work on agent-based product-line CBSE for flexible e-commerce systems. We are integrating several technologies for product-line analysis and component design, implementation and customization to create a basis for systematic product-line development. Largely independent work on reuse ("domain analysis") and object-oriented technology ("design and code") has matured to the degree that integration of the techniques promises a coherent approach. We outline a practical development process that structures a set of common and variable features supporting a product-line, to produce reusable elements ("aspects") that can be combined into customized components and frameworks to support the product-line.

1. INTRODUCTION

1.1 Web-based E-Commerce

Web-based E-commerce is becoming more pervasive each day. The increasing volume of Business-to-Consumer (B2C) and Business-to-Business (B2B) Internet traffic provides great opportunity for automation (Sharma, 1999; Glushko*, et al.*, 1999). Information search, analysis and negotiation will be done by automated assistants. These new classes of application demand highly flexible, intelligent solutions. Developers need more powerful ways of quickly building these many related applications and supporting services. Component-based software engineering (CBSE), product-line development and agent technology offer significant promise.

Software agents can be viewed as a specialized kind of distributed component, offering greater flexibility than traditional components. There are many kinds of software agent, with differing features such as mobility, autonomy, collaboration, persistence and intelligence. Effective exploitation of this variety consistently across a set of agents is essential to benefiting from the agent-oriented paradigm. At HP Laboratories, we are using product-line CBSE approaches to analyze the commonality and variability within these agent systems, and to express this commonality and variability effectively using component composition and generation technology.

In the following sections, we will summarize product-lines, agents and feature-driven domain analysis. We then describe how the commonality and variability defined by an explicit feature model can be expressed in customized product-line components by compositional and generative aspect-oriented technologies. We conclude with an outline of our feature-driven, aspect-oriented product-line method, and some examples.

1.2 Product-lines and component-based software engineering

There are great benefits to managing related products and applications as members of a product-line. Systematic, architected 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. Product-lines enable more effective component reuse. 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.

Many researchers are working on these technologies from multiple perspectives, leading to a somewhat confusing and overlapping terminology. In a companion paper (Griss, 2000a), we carefully define many of the terms used in this paper, and propose steps towards unification.

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 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 standards compliance.

For example, each different application in an agent-based e-commerce product-line might optionally provide payment using credit cards and/or e-cash, allow the deployment of customer profile-based shopping agents, provide one or more kinds of comparison shopper or product recommender agents, empower autonomous shopping and auction agents, and permit e-mail, pager and/or telephone alerts, if autonomous buying agents are also deployed. Some agents may accept voice input; some may run in any web browser, while some depend critically on distinct features of a particular Windows 2000, HP-UX, or Linux platform.

This commonality and variability can be exploited by treating the product-line as a family and decomposing into shared reusable components that separate concerns (Parnas, 1972, 1976, 1979). The approach first analyzes the products to determine common and variable features, and then develops a product structure and implementation strategy that expresses this commonality in terms of reusable components.

For example, (Bosch, 1999) 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. Similarly, the HP Laboratories component framework (Griss, 2000a) for medical and financial applications generates a customized C++ or Java framework for each distinct class of component. Components are then completed and specialized by subclassing or adding other code. The framework is characterized by a cluster of features, and the added code and selections of optional code and parameter settings correspond to the variable features.

Each component captures some of the features, or allows features to be built up by combining components. For example, each member of an agent-based product-line might share a common XML-based communication component, and optionally one or more components for payment, customer profile management, or workflow-based collaboration. These components

may be combined to produce a variety of similar agent systems, only some of which would be viable e-commerce products. Some components are used without change in plug-compatible sets; other components are adapted or parameterized to account for product differences; and yet other components are substantially modified before inclusion into the product.

1.3 Agent-based E-commerce systems

Our agent-based e-commerce research at HP Laboratories focuses on agent communication, agent-based systems management and control, mobility, and multi-agent collaboration (Griss & Kessler, 1996; Griss, 2000; Chen, *et al.*, 2000; Machiraju, *et al.,* 2000). We are interested in how agent components are quickly defined and constructed, how they communicate, and how collaborations between groups of agents can be established and controlled.

Most people have an informal sense of what agents are - intelligent pieces of software that operate somewhat autonomously to represent a user's interests. There are many books about agents (Huhns & Singh, 1998; Jennings & Wooldridge, 1998; Bradshaw, 1997), as well as numerous papers, and web sites.

There are over 100 published agent systems, each providing and emphasizing different combinations of features and implementations. The major categories of agents include **personal agents** which interact directly with the user and are highly personalized, **mobile agents** which are sent out to collect information or perform actions at one or more remote sites and then return with results, and **collaborative** or **social agents** which communicate and interact within a multi-agent system.

We view agents as a special kind of distributed software component. The benefits and process of working with agents adds to those of working with components - modular, independently developed pieces of software (components) can be combined in a variety of ways to produce new application systems.

A typical agent system, such as Zeus (Nwana, *et al.,* 1999), provides different amounts and models of security, communication, conversation control, persistence and mobility. An agent platform manages creation, deletion, monitoring, authentication and migration of agents. It routes agent messages and provides naming and directory service to locate agents by name and by function. Some services can also be provided as special agents, such as Name Servers and Facilitators.

Our current agents are built using Java, XML and HTTP; others use (D)COM, C++, Visual Basic Script or TCL. Different capabilities and implementations are used for particular multi-agent systems. For example,

agent messaging could be implemented using XML structured documents described by DTD's or XML Schema, with some agent parameters passed in HTTP headers. Alternatively, TCP/IP sockets or UDP and a standard LISP-structured agent communication language (KQML or FIPA ACL) can be used. Naming requires system wide, consistent choices to be made. Agents may have a globally unique name (e.g. GUID or URN), or they may have one or more local names based on their host machine or agent group. Likewise, agent conversation management must be handled compatibly across agents. Loose control assigns to each agent a set of rules that control conversations; tighter control expects compatible agents to use a common protocol (e.g. state machine representation and engine), while the tightest control will associate a single collaboration workflow model and engine with a group of agents.

A component can be usefully viewed as an agent component when some amount of mobility, intelligence, autonomy, adaptability, collaboration and personality are present (Griss, 2000). Some of these properties may be selected almost independently, but several must be selected consistently to make agents work together as a flexible yet robust whole. Experience with this "almost orthogonal feature" decomposition of agents motivates a feature-driven, aspect-oriented implementation.

1.4 Domain analysis helps find features

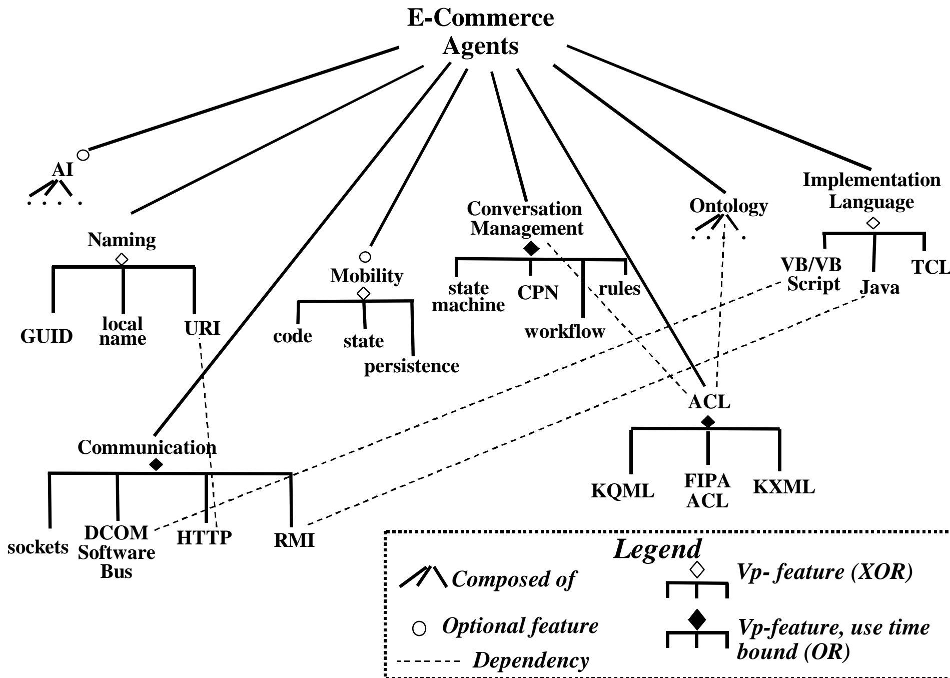
Domain analysis or domain engineering techniques (Arango, 1994) are used to systematically extract features from existing or planned members of a product-line. Feature trees are used to relate features to each other in various ways, showing sub-features, alternative features, optional features, dependent features or conflicting features. These methods, such as FODA (Kang, 1990), FORM (Kang *et al.,* 1998,1998a, 1999), FeatuRSEB (Griss, *et al.,* 1998), FAST, ODM, and Pulse are used to identify and 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.

FORM (feature-oriented reuse method) is an extension to FODA (feature-oriented domain analysis) that 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.

FeatuRSEB augments the use-case driven RSEB method (Jacobson, *et al.,* 1997) with an explicit FODA-based feature model to provide a feature index into common and variable use-case, design and implementation elements.

Figure 1 shows an example of a FeaturRSEB feature tree diagram for parts of our agent component system. Solid lines show commonality and variability structure, while the dashed lines show some "non-orthogonal" dependencies.

Once each component is designed, it is assigned to a component development team which develops and maintains it. Complete products are assigned to application development (or "feature" or "reuser") 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.

Figure 1. Simplified feature tree for some of the choices in implementing agent components.

1.5 Mapping cross-cutting features to components is complex

This feature-driven CBSE approach works well in many (ideal) cases, when a particular feature will be implemented in code that is mostly localized to a single module. This occurs when the features decompose into a set of almost independent clusters, each cluster implemented by fixed

components. Very few features span multiple components, leading to "almost orthogonal separation of concerns."

Things become more complex when components have compile-time or run-time parameters that customize components. Developers must track which parameter settings relate to which versions of components, and to which set of feature selections. This gets very complicated when the code implementing a particular feature or closely related set of features is scattered across multiple components, classes or modules. It gets extremely complicated when this code is intertwined with code implementing other (groups of) features, cutting across multiple components. Such "crosscutting" concerns result in great difficulty in associating separate concerns with separate components.

Example crosscutting features include middleware concerns such as end-to-end performance, transaction or security guarantees, or agent capability concerns such as functional coherence, mobility model, naming model consistent with mobility choice, agent communication and conversation control. Other cross-cutting e-commerce agent system architecture concerns involve distribution and scalability architecture, high-availability, replication strategy, binding of application servers and agencies, and federation of name servers and facilitators.

A key observation is that individual features (or small sets of related features) typically 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. Particularly in the case of e-commerce systems, the ability to rapidly and consistently evolve (sets of related) features is key.

This is the same observation that makes usecase-driven development so powerful; it is complete usecases that change or are selected almost independently. In the RSEB (Jacobson*, et al.*, 1997) and FeatuRSEB approaches (Griss *et al.*, 1998) to product-line development, we address usecases separately and choose among variant usecase features. 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 several design and implementation components.

In the case of significant crosscutting variability, maintenance and evolution becomes very complicated. Individual features will then not 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" feature-interaction problem is closely related the general requirements traceability problem.

Separation of concerns is thus most effective at the higher (use-case or feature) level, before "tangling" and "crosscutting" into design or implementation.

2. UNTANGLING THE FEATURE WEB

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

Traceability - Use hyperweb-like tools to help build and maintain fairly complex maps ("hyper-webs") from features to products, and manage the evolution of features by jointly managing the evolution of the many related components. Powerful requirements tracing and management tools have made it somewhat easier to find and manage slices corresponding to one feature. See for example the DOORS or Calibre RM requirements management tools, and UML language and tool-based traceability (Booch*, et al.*, 1999). In some cases, the trace from requirements through features to design and code are maintained as the software is built; in others, the traces are discovered using maintenance and analysis tools. Even with such traceability tools, when one feature changes, code in many modules will have to change and multiple versions maintained. Not only is it hard to find and disentangle the code corresponding to one feature from the code for other features, but since lots of code in many components is touched, many components have to be retested and revalidated after a change.

Architecture - Use tools and techniques such as patterns (Gamma*, et al.*, 1994; Buschman*, et al.*, 1996), styles (Garlan & Shaw, 1996), and layering and decoupling technologies such software buses and middleware to provide a good decomposition into separate, orthogonal pieces. This leads to a better structured, more maintainable 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.

Composition or "weaving" of program fragments ("aspects") - Use language extensions or preprocessors to convert features into software fragments ("aspects") that can be assembled into complete classes, components or products. Previously a systematic approach to aspects has been lacking. Earlier techniques used parameters, macros, classes, inheritance and templates in an *ad hoc* manner. Macros create inline code, overcoming a concern about performance when breaking a system into many

small procedures. Macros can also generate alternative pieces of code using conditional expansion, testing compile-time parameters.

Note that the generative, aspect-oriented techniques we describe in the remainder of the paper do not obviate the need to maintain traceability from requirements and features to components and aspects. That is, the feature diagram is not maintained as an isolated model, but should be dynamically linked and traceable to other model elements (Jacobson *et al.,* 1997; Griss *et al.,* 1998; Booch *et al.,* 1999, Tarr *et al*, 1999).

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

A systematic approach to combining feature-driven and aspect-oriented techniques is emerging. We use the term *feature-driven* to refer to the techniques in the front of the process, namely, the identification and analysis of feature commonality and variability, as well as architecting and some design. We use the term *aspect-oriented* to refer to the back of the process, namely, the identification, design, construction and management of design and implementation fragments and components.

The essential idea is to directly implement a feature as a fragment of code (an *aspect*), and then use some mechanism or tool to combine these fragments into complete methods, classes or components. The goal is not to directly manage the resulting components, but to only manage features and corresponding aspects. When needed, a new version of a complete component is (re-)generated.

Several different techniques and tools have been developed to make "composition and weaving" practical. These differ primarily in the granularity and language level of aspect fragments and the legal ways in which they can be combined. For example, code could be woven at the class level, at the method level or within individual statements. The approach could be seen as a just a disciplined way of using a standard language feature such as C++ templates or inheritance, or could supported by a tool that effectively changes the language such as a preprocessor or language extension.

In a companion paper (Griss, 2000a), we provide a brief history of the key developments underlying these compositional and generative approaches and their evolution. There have been numerous efforts to design technology that separates concerns and makes the specification and implementation of layered abstractions easier. Notable examples are parameterized programming (Goguen, 1984), frameworks (Johnson & Foote, 1988), meta-programming (Kiczales *et al.,* 1991), generic programming (Musser &

Stepanov, 1994; Musser & Saini, 1996; Dehnert & Stepanov, 1998; Jazayeri, 1998), system generators (Batory & O'Malley, 1992), role and layer mixins (Van Hilst & Notkin, 1996, 1996a, 1996b; Smaragdakis & Batory, 1998, 1998a) and C++ template meta programming (Czarnecki & Eisenecker, 1998,1999). Recent work takes a more systematic approach to identifying and representing crosscutting concerns and component composition.

We will discuss only a few examples to illustrate the range of techniques and not to advocate any particular technique over others.

OO inheritance and frameworks - Using (C++) frameworks (Johnson & Foote, 1988), subclasses define new methods that "override" and augment the definition of some default methods inherited from super-classes. The framework acts as a skeleton into which class-sized aspects are "composed." It is worth noting that many design patterns use inheritance and delegation to help encapsulate some crosscutting features (Gamma, *et al.*,1994; Buschmann, *et al.*,1996).

C++ templates, generic programming and "mixins" - There are several distinct models (or paradigms) for using OO (C++) templates for finer-grained composition- parameterized data types or generics without inheritance, layer and role mixins, and parameterized algorithms and/or compile time computation (e.g. for tailoring matrix algorithms). The body of a method or several methods is created by composing C++ code fragments using templates.

C++ templates were first heavily exploited in the C++ Standard Template Library (STL), in which a class is constructed from a composition of an Algorithm, a Container, an Element, and several Adapter elements (Musser & Saini, 1996; Musser & Stepanov 1994; Dehnert & Stepanov, 1998). STL is noteworthy for its lack of inheritance. VanHilst and Notkin (Van Hilst & Notkin, 1996, 1996a, 1996b) use templates to implement role "mixins" for expressing a collaboration between several OO-components. By using multiple-inheritance or layering of "mixin" classes, subclasses become 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.

Frame-based generators - Basset uses a simple frame-based "generator" (called an "assembler") to compose code fragments from several frames, linked into a form of "inheritance" or "delegation" hierarchy (Bassett, 1996). 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 *ad hoc* frame structures may be needed to achieve quite complex code transformation and generation, a good

decomposition has each frame dealing with only one or a small number of distinct aspects. Frame technology has been used with impressive results by Netron and its clients to implement several product lines rapidly and efficiently. We have also used a frame generator in our customizable component framework (Griss, 2000a).

Subject-oriented programming (SOP) - Ossher, Harrison and colleagues (Harrison & Ossher, 1993; Walker*, et al.*, 1999; Clarke*, et al.*, 1999) represent each concern or feature as a separate package, implemented or designed separately. SOP realizes a model of object-oriented code whose classes, methods and variables are independently selected, matched and finally combined to produce the complete program. Systems build up as a composition of subjects as 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 as C++ 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*,* (Clarke, *et al.*, 1999).

Aspect-oriented programming (AOP) - Developed by Kiczales and colleagues (Kiczales*, et al.*, 1997; Lopes & Kiczales, 1998), AOP expands on SOP concepts by providing modular support for direct programming of crosscutting concerns (Walker*, et al.*,1999). 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. One or more aspects (which are largely orthogonal if well designed) are applied to components, performing 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 in the base code at which insertions or modifications may occur, similar to RSEB variation points. Join points may be as generic as constructs in the host programming language or as specific as application-specific event or code patterns. One such language, AspectJ (Lopes & Kiczales, 1998), 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, refining, modifying and extending a relatively complete program into a newer program.

System generators - Batory's GenVoca and P++ (Batory & O'Malley, 1992; Batory, 1996) uses a specialized preprocessor to assemble or compose systems from layered specifications. GenVoca eschews the arbitrary code transformations and refinements possible with AOP and SOP, instead carefully defining (e.g., with a typing expression) and composing layered abstractions, and then validating compositions. GenVoca has been applied to customizable databases, compilers, data structure libraries and other domains. Weaving originally done with a language such as P++ has been recently replaced by a clever combination of nested C++ templates and inheritance. Batory and Smaragdakis simplified and extended role mixin idea (Van Hilst & Notkin) into mixin layers (Smaragdakis & Batory, 1998, 1998a; Smaragdakis, 1999). Using parameterized sets of classes and templates, groups of application classes are built up from layers of super-classes that can be easily composed. They show how P++ and GenVoca style composition can use C++ template mixin layers, instead of a special generator.

As in RSEB and FeatureRSEB, these mixin techniques express an application as a layered composition of independently defined collaborations. 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 gives rise to separate code that must then be woven into the complete system. The mixin-layer 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 obscure manner.

Related work has been done by Lieberherr and colleagues (Liberherr & Xiao, 1993; Mezini & Lieberherr, 1998) on Demeter's flexible composition of program fragments, and by Aksit (Aksit, *et al.*, 1992; Aksit, 1998) and others.

Hyperwebs and traceability combined with aspects - Tarr (Tarr, *et al.*, 1999) shows how separation of concerns using SOP or AOP leads to a hyperweb tracing from specific concerns through parts of multiple components. Each "hyperslice" through this web corresponds to a different composition/decomposition of the application. There are a variety of different ways to decompose the system, and this stresses any particular decomposition strategy. This is the "tyranny of the dominant decomposition" - no single decomposition, including features, is always adequate, as developers may have other concerns instead/in addition to features (or functions, or classes, or whatever). Once a dominant decomposition of the software is chosen, all subsequent additions and modifications corresponding to a particular aspect will have to be structured correspondingly. Thus, in some cases, what is conceptually an orthogonal aspect will result in some

code fragments that will be aware of the existence and structure of other aspects. See also Van Hilst's discussion of the Parnas' KWIK example (Van Hilst & Notkin, 1996b).

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 equivalent to model element traceability across models in UML (Booch*, et al.*, 1999), RSEB (Jacobson, *et al.,* 1997) and FeatuRSEB (Griss, *et al.,* 1998).

4. FEATURE-DRIVEN, ASPECT-ORIENTED PRODUCT-LINE CBSE

A systematic approach to feature-driven, aspect-oriented product line development with components is thus quite simple in concept:

1. Use a feature-driven analysis and design method, such as FeatuRSEB, to develop a feature model and high-level architecture and design with explicit variability and traceability. The feature model is developed in parallel with other models as an index into the product-line family usecases. Features are also traced to collaborations, variation points and variants in the design and implementation models (Griss, *et al.,* 1998). Design and implementation models will thus also show 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 combinations.
3. Express the aspects as code fragments 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 from corresponding fragments.

5. EXAMPLES AND EXPERIENCE

At Hewlett-Packard Laboratories we are exploring these techniques in the context of two major classes of application prototypes. As described in (Griss, 2000a) we have prototyped several large-scale domain-specific component frameworks for families of medical and banking applications. We used a simple FeatureRSEB-like usecase-based domain analysis to analyze variability. Components have multiple interfaces, several of which support compatible crosscutting features. A framework for each component

is generated using a Basset-style frame generator, and then manually completed. Frames are used to represent different aspects, such as security, workflow/business process interfaces, transactions, sessions and component lifecycle.

Our current focus is the development of prototype agent based systems for e-commerce and service management (Griss, 2000; Machiraju, *et al.*, 2000). We are developing a variety of different, yet compatible agents. Each of these agents will have a different mix of the features illustrated in Figure 1. Changing the details of one feature (such as agent communication language, mobility, or conversation management strategy) will require compatible changes across many agents. As more agents are explored and deployed, it will be increasingly important to make it easier to define and implement different agent systems directly in terms of their features. Consistent with Figure 1, this suggests that agents be constructed by combining several aspects.

In our recent work on management agents (Machiraju, *et al.*, 2000; Shahoumian, 2000), we construct task-specific agents out of lower-level "parts" implemented as COM components, connected by an event bus. The specific parts, their software-bus interconnections, and their customizing parameters are loaded from a management model to dynamically assemble an agent as it is loaded into an agency on a target machine. The model and then code is constructed by a visual or textual generator. This is similar to the visual generative techniques used to compose task- and platform-customized agents in Zeus (Nwana, *et al.*, 1999), for mobile information agents (Falchuk & Karmouch, 1998), and for mobile for management agents using the AgentBean Development Kit (ADK) (Gschwind, *et al.*, 1999).

ADK builds families of mobile management agents by composing several components (Java Beans). Each class of component handles a key agent aspect, such as mobile agent navigation and touring, domain-specific actions to be performed at each place visited, and reporting of results to other agents or users. ADK components are related by several cross-cutting patterns, ensuring consistency of event-based communication, navigation, actions and reporting. Rather than using a specialized generator, ADK uses Sun's BeanBox as a "visual builder/generator," composing a complete agent component for lower level aspect components. ADK provides a higher-level abstraction layer for implementing domain-specific solutions using several different mobile agent platforms. A stated limitation of Java Beans and the BeanBox is that the typed event model and static interfaces are too constraining, and will likely require the use of a more powerful aspect-oriented composition tool or generator to manage the variety of cross-cutting concerns.

Finally, Kendall discusses the use of aspects to implement agent role models (Kendall, 1999). A set of agents collaborating to accomplish some task, perhaps following some process steps in a workflow, play distinct roles, such as Manager, Supervisor, and Clerk, or Buyer, Seller, and Auctioneer. A single agent might play multiple roles, in different situations or even at the same time. A (UML) role model is used to describe the interactions between the agents as a cross-cutting pattern. It maps the agent responsibilities onto several collaborating classes that implement the agents. Each role defines a compatible set of responsibilities that must be distributed across multiple objects, and several roles provide parts of the complete set of responsibilities that must be composed into a single class. In some cases, inheritance and templates will suffice to manage the "object schizophrenia," but in many others, such as for dynamically changing roles, an aspect-oriented implementation is needed.

6. SUMMARY AND CONCLUSIONS

We have provided an overview of the overall approach and some details of a systematic, feature-driven, aspect-oriented component-based product-line development approach.

As more agent-based systems are explored and deployed for E-Commerce, it will be critical to make it easier to define and implement these product-lines by composing basic agent capabilities into consistent, interoperable system. Agents will be constructed by combining several aspects, representing key features, cross-cutting agent system concerns and role-based workflow coordination.

Aspect-based development is not only limited to smaller products, but can be used for the customization of the components of large products. Highly-customized, fine-grained aspect-based development is complementary to large-grain component-based development. In RSEB and FeatuRSEB we view aspect code fragments implementing specific features as a type of (fine-grain) component, just as we view C++ templates or frames. In RSEB and FeatuRSEB, we focus on large-grain, customizable components and frameworks (*aka* Component Systems), in which variant aspects are "attached" at "variation points."

Feature-driven domain analysis and design and aspect-oriented implementation techniques have matured to the point that it seems possible to create a practical path for product-line CBSE. Starting from an analysis of 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. This simple prescription is not yet a complete method, but suggests that product-line architects, domain engineers and component and framework designers should employ feature-driven and aspect-oriented techniques to better structure their models, designs and code.

ACKNOWLEDGEMENTS

I am grateful for several useful and extremely clarifying suggestions and comments on early drafts of this paper by Don Batory, David Bell, Jon Gustafson, Gregor Kiczales, Chris Preist, Peri Tarr and Michael Van Hilst.

REFERENCES

Aksit, M. (1996), Composition and Separation of Concerns in the Object-Oriented Model, in *ACM Computing Surveys* 28A(4), December.

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

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

Bassett, P.G. (1996), *Framing Reuse: Lessons from the Real World,* Prentice Hall.

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

Batory, D. and O'Malley, S. (1992), The design and implementation of hierarchical software systems with reusable components, *ACM Transactions on Software Engineering and Methodology*, 1(4):355-398.

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

Bosch, J. (1999), Product-Line Architectures and Industry: A Case Study, in *Proceedings of ICSE 99,* 16-22 May, Los Angeles, California, USA, ACM press, pp. 544-554.

Bradshaw, J.M (1997), *Software Agents*, MIT Press.

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

Chen, Q., Hsu, M., Dayal, U., and Griss, M. (2000), Multi-Agent Cooperation, Dynamic Workflow and XML for E-Commerce Automation, in *Proc. Autonomous Agents 2000*, June, Barcelona.

Clarke, S., Harrison, W., Ossher, H. and Tarr, P. (1999), Towards Improved Alignment of Requirements, Design and Code, in *Proceedings of OOPSLA* 1999, ACM , pp. 325-339.

Czarnecki, K. and Eisenecker, U.W. (1999), Components and Generative Programming, in *Proc. ACM SIGSOFT 1999* (ESEC/FSE), LNCS 1687, Springer-Verlag. (See also Available at *http://home.t-online.de/home/Ulrich.Eisenecker/meta.htm*)

Dehnert, J. and Stepanov A.A. (1998), Fundamentals of Generic Programming, *Proc. Dagstuhl Seminar on Generic Programming,* April 27--May 1, Schloß Dagstuhl, Wadern, Germany.

Falchuk, B. and Karmouch, A. (1998), Visual Modeling for Agent-Based Applications, *IEEE Computer*, Vol. 31, No. 12, December. Pp. 31 - 37.

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

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

Glushko, R. J., Tenenbaum, J.M. and Meltzer, B. (1999), An XML framework for agent-based E-commerce. *Communications of the ACM*, Vol.42, March.

Goguen, J. A. (1984), Parameterized Programming, *IEEE Trans. Software Eng.*, SE-10(5), September, pp. 528-543.

Griss, M. L. (2000), My Agent Will Call Your Agent, *Software Development Magazine*, February. (See also "My Agent Will Call Your Agent … But Will It Respond?" Hewlett-Packard Laboratories, *Technical Report, TR-HPL- 1999-159*).

Griss, M. L. (2000a), Implementing Product-Line Features with Component Reuse, in *Proceedings of 6th International Conference on Software Reuse*, Springer-Verlag, Vienna, Austria, June.

Griss, M. L. and Kessler, R. R. (1996), Building Object-Oriented Instrument Kits, *Object Magazine*, April.

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

Gschwind, T., Feridun, M. and Pleisch, S. (1999), ADK - Building Mobile Agents for Network and Systems Management from Reusable Components, in *Proc. of ASA/MA 99*, Oct, Palm Springs, CA, IEEE-CS, pp 13-21. (See http://www.infosys.tuwien.ac.at/ADK).

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

Huhns, M. N. and Singh, M. P. (1998), *Readings in Agents*, Morgan-Kaufman.

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

Jazayeri, M. (1998), Evaluating Generic Programming in Practice, *Proc. Dagstuhl Seminar on Generic Programming,* April 27--May 1, Schloß Dagstuhl, Wadern, Germany.

Jennings, N. R. and Wooldridge, M. R. (1998), *Agent Technology,* Springer.

Johnson, R. and Foote, B. (1988), Designing reusable classes, *JOOP*, pp. 22-30, 35, June.

Kang, K. C., (1998a), Feature-oriented Development of Applications for a Domain*, Proceedings of 5th International Conference on Software Reuse*, June 2 - 5. Victoria, British Columbia, Canada. IEEE , pp. 354-355.

Kang, K. C., *et al.*, (1990), Feature-Oriented Domain Analysis Feasibility Study, SEI Technical Report CMU/SEI-90-TR-21, November.

Kang, K. C., *et al.*, (1998), FORM: A feature-oriented reuse method with domain-specific architectures, *Annals of Software Engineering*, V5, pp. 143-168.

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

Kendall, E. A. (1999), Role Model Designs and Implementations with Aspect-oriented Programming, in *Proc. of OOPSLA 99*, Oct, Denver, Co., ACM SIGPLAN, 353- 369.

Kiczales, G., des Rivières, J. and Bobrow, D. G.(1991), *The Art of the Metaobject Protocol,* MIT Press.

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

Lieberherr, K. J. and Xiao, C. (1993), Object-oriented Software Evolution*, IEEE Transactions on Software Engineering*, v 19, No. 4, pp. 313-343, April.

Lopes, CV and Kiczales, G. (1998), Recent Developments in AspectJ™**,** In *ECOOP'98 Workshop Reader*, Springer-Verlag LNCS 1543.

Machiraju, V., Dekhil, M., Wurster, K., Garg, P., Griss, M. and Holland, J. (1999), Towards Generic Application Auto-discovery, 2000 IEEE/IFIP Network and Operations Management Symposium, Honolulu, Hawaii, April 2000. (See also Hewlett-Packard Laboratories Technical Report HPL-1999-80).

Mezini, M. and Lieberherr, K. (1998), Adaptive Plug-and-Play Components for Evolutionary Software Development, *Proceedings of OOPSLA'98,* pp. 97-116.

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

Musser, D. R. and Stepanov, A. A. (1994), Algorithm-Oriented Generic Libraries. In *Software Practice and Experience*, Vol 24(7).

Nwana, H., Nduma, D., Lee, L., Collis, J. (1999), ZEUS: a toolkit for building distributed multi-agent systems, in *Artificial Intelligence Journal*, Vol. 13, No. 1, pp. 129-186.

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

Parnas, D. L. (1979), Designing Software for Ease of Extension and Contraction*, IEEE Transactions on Software Engineering*, v 5, #6, pp 310-320, March.

Parnas, D. L., (1972), On the criteria to be used in decomposing systems into modules, *CACM*, 15(12):1053-1058, December.

Shahoumian, T. A. (2000), A Specialized Macro Language For Specifying the Communication Patterns Within an Agent, *Hewlett-Packard Laboratories Technical Report,* HPL-2000-07.

Sharma, T. (1999), E-Commerce Components, *Software Development*, Vol 7, August.

Smaragdakis, Y. (1999), Reusable Object-Oriented Components, *Proceedings of Ninth Annual Workshop on Software Reuse,* Jan. 7-9, 1999, Austin, Texas.

Smaragdakis, Y. and Batory, D. (1998a), Implementing Layered Designs with Mixin Layers*, Proc. of ECOOP98.*

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

Tarr, P. Ossher, H., Harrison, W. and Sutton, Jr. S. M., (1999), N degrees of Separation: Multi-dimensional separation of concerns, *Proc. ICSE 99,* IEEE, Las Angeles, May, ACM press, pp. 107-119.

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

Van Hilst, M. and Notkin, D. (1996a), Using Role Components to Implement Collaboration-Based Designs*, Proc. OOPSLA96,* 359-369.

Van Hilst, M. and Notkin, D. (1996b), Decoupling Change From Design*, Proc. ACM SIGSOFT 1996.*

Walker, R, J., Banniassad, E. L. A. and Murphy, G. C. (1999), An Initial Assessment of Aspect Oriented Programming, *Proc. ICSE 99*, IEEE, Las Angeles, May, pp. 120-130.