CBSE Success Factors: Integrating Architecture, Process, and Organization

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

(*Chapter 9 in* ***Component-Based Software Engineering: Putting the Pieces Together,*** *Edited by George T. Heineman, Ph.D. & William Councill, M.S., J.D., May 2001, Addison-Wesley*

Introduction

A *product-line* is a set of applications that share a common set of requirements but also exhibit significant variability in requirements. CBSE can exploit this commonality and variability to reduce overall development costs and time. (See chapters 15, 22, and 36 for more on product lines).

Important connections exist among product-line CBSE, systematic reuse, component infrastructure, and the processes and organization that produce a product-line. Most organizations successfully adopt CBSE incrementally by carefully matching the new technology with a business need and organizational process maturity. To effectively develop a product-line, you need a coherent approach to architecting the system, to designing and structuring the components and component infrastructure, to organizing the workforce of architects, designers and implementers, and to convert the development and business processes to CBSE-based methods.

My approach has been shaped by business process engineering, UML modeling, and the SEI process maturity frameworks. This chapter draws on my book "*Software Reuse: Architecture, Process and Organization for Business and Success*"(Jacobson, Griss and Jonsson, 1997) and recent articles (Griss 1998, 1999, 2000). A reuse-driven software engineering business (RSEB) is a software development organization that practices large-scale component-based product-line development and systematic reuse. Business (process) engineering and model-driven techniques provide a comprehensive and systematic approach to orchestrate the large-scale investment and change needed to establish an effective component-based reuse program. In some cases, optimizations and the specific situation may allow or require that some steps be omitted or modified.

Obstacles to Effective Component Reuse

Many organizations engage in informal reuse through code sharing or design patterns. However, systematic reuse of software components across multiple applications and projects remains in its infancy. You will face many obstacles as you make the transition from traditional software development to component-based enterprise software development (Frakes 1994). To overcome these obstacles, a variety of issues must be addressed (Jacobson, *et al*, 1997; Favaro *et. al* 1998; Pour 1998; Bosch 1999):

1. ***Business***: how component development, support and training should be funded; lack of access to vendor-supplied components; lack of a convincing business case and economic model for long term investment; and, unclear definition of product-line.
2. ***Process***: low process maturity of the organization; ill-defined or unfamiliar reuse-oriented methods and processes; new coordination and management needs; and, absence of well-tested and documented methods and models to relate features to component sets and variability.

3. ***Organization***: lack of a systematic practice for reuse activities and enterprise component development; lack of management expertise and support; and, cultural and trust conflicts.
4. ***Engineering***: lack of adequate techniques and tools for identifying, designing, documenting, testing, packaging, and categorizing reusable software components; too few and poorly understood standard patterns and architectures.
5. ***Infrastructure***: lack of widespread use of a standardized design notation such as the UML; lack of tools and components; too many different programming languages and environments; and, lack of support for multi-group configuration management.

Companies must make numerous decisions as they develop software-intensive products for rapidly moving markets, such as Web-enabled applications or e-commerce systems (Griss 1997, 1998). These involve:

1. ***Time*****:** Rapid product development and market agility are critical when developing distributed software products. How should these time pressures affect processes, new architectures and component infrastructures, business measures and organizational structures?
2. ***Process*****:** How to adjust your process to move from a strictly feature-driven to a reuse-driven process? . This means that instead of planning releases based only on the features they deliver, you increase priority of those features that can be delivered by reusable elements. What standard processes and process maturity levels (such as the SEI CMM) should be selected? How do we achieve widespread use in the most expeditious way? How do these relate to reuse and CBSE?
3. ***Organization*****:** There may be cultural and organizational issues that impede effective ways of working with architected systems and components. Who owns standards, architectures and component infrastructures? Who pays for component development? What discourages component sharing? What organizational, management, and measurement changes are needed to encourage change?
4. ***Coordination*****:** Coordination between new and ongoing technical and process improvement initiatives is needed to avoid redundant or conflicting efforts. What are the connections between new technologies and strategies for component development, improving process predictability, improving quality and decreasing cycle time? What are the priorities? What standards are needed?
5. ***Technology*****:** Common architecture, patterns and component infrastructures, reusable components, and leverageable platforms are key to achieving decreased cycle time. What notations, standards, and technologies should be used? How should they be introduced? What about standard tools?

Figure 1: Critical Success Factors for Product Line CBSE

Business-driven product-line

- The organization must have a visible, articulated and compelling need for dramatic improvements in cycle time, cost, productivity, agility and/or interoperability.
- The organization must produce software (applications, embedded systems or key components) that form an obvious product-line, application family or coherent "domain."

Process-oriented

- Distinct software development and maintenance processes for architecture and component infrastructures, components, and applications must be defined and followed.
- These processes must explicitly incorporate reuse. They systematically identify and express commonality and variability, and include standards for designing and packaging components for reuse.

Architected

- Applications, systems and components must be purposefully designed and structured to ensure that they fit together, and that they will cover the needs of the family or domain.
- A well defined, layered, modular structure, and various design and implementation standards (such as the UML and patterns catalogs) will be of great help.

Organized

- Long-term management commitment is needed to ensure that the organization is structured, trained, and staffed to follow component reuse processes and conform to standards.
- Distinct subgroups are needed to create, reuse, and support reusable components. These teams must be trained to follow the specific processes.
- People must have well defined reuse-oriented jobs, with appropriate training, skills and rewards for effective reuse performance.

Critical Success Factors for Product-Line CBSE

You need a systematic approach to coordinate the management and development efforts and resources effectively. In my work at Hewlett Packard and study of Ericsson, Rational, Intecs Sistemi and several HP customers, I have found that a successful, large-scale component reuse effort must be coherent and holistic. Foremost, the component-based software reuse program must be aligned with, and driven by, a compelling business reason, such as the critical need to decrease time-to-market or to meet competitive market forces. Business needs and product-line development provide a context of organizational commitment in which CBSE can be justified economically, technically and strategically. This will then enable process and technical change.

An effective product-line CBSE program has the critical success factors shown in sidebar, figure 1ve. Because of the magnitude of the changes and the careful orchestration needed, I advocate a business process engineering strategy to restructure a software engineering organization for large-scale reuse (Griss 1995). The processes are built on Jacobson's use case-driven methods (Jacobson *et. al.,* 1992, 1994) to produce the reuse-driven software engineering business (RSEB), an integrated, model-driven approach.

Integrating Architecture, Process, and Organization

A RSEB is run as a software engineering business. Software engineering goals are key to accomplishing the organization's business goals, and as a consequence, the software organization itself must be operated with compatible customer and financial objectives. All processes and work products should be aligned to

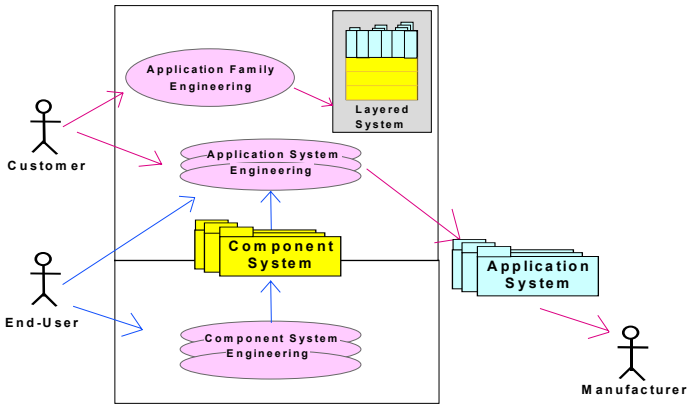

Figure 2: The Elements of the Reuse-Driven Software Engineering Business

these business goals. For example, in several HP divisions, a dominant and compelling business goal is to reduce product development times, yet retain market agility across product families. Such families are conceived to meet different customer and country needs by combining reusable components. For example a group of divisions at HP (now at Agilent) , builds microwave instruments from common firmware components ito create a family of compatible test systems that are configured to a variety of situations. One cross-divisional team was set up to craft the architecture for the family and design the initial components. Other groups within the divisions created components consistent with this architecture, or developed applications using them. A final group was established to support and maintain the components. To initiate and coordinate the efforts of these several divisions required involvement of senior management.

Senior management must make a strategic decision to establish one or more reuse-driven business units, and create a context in which they will work together These units will produce multiple, related applications, optimized around the production and reuse of components, forming an explicit component-based product-line. An organization will only change because of an appropriate level of management commitment, for example, a strategic statement from senior management (i.e. improve the speed with

which new products are developed) and an adequate long-term budget. Business tradeoffs, such as expense vs. time to market vs. profit must be managed using well-defined economic, product, and process measures.

Model-driven Development using a Standard Modeling Language

Figure 2 summarizes the key elements of the RSEB using the Unified Modeling Language (UML) Booch, et al. 1999), an established OMG standard modeling language : The ellipses are business use cases, representing software engineering processes, while the tabbed rectangles are systems, representing sets of UML model elements. The stick figures are actors, representing people or organizations with which the RSEB interacts.

Each system in Figure 2is expressed as a set of UML models.. The UML provides software designers an unprecedented opportunity to develop and reuse precise and widely understood blueprints for software designs and reusable infrastructures. As described below, we use the same notation to model the RSEB processes.

Architecture

While I have tried to adjust my terminology to that of Chapter 1 and 3, there remain some inevitable inconsistencies because the field of CBSE is still immature. There is a growing awareness in the Software Engineering community on the importance of *Software Architecture* (Jacobson *et al.,* 1997; Griss *et al.,* 1998, Garlan and Shaw, 1996, Kruchten, 1995;Buschmann, *et al.* 1996; Mowbray and Malveau, 1997). *Architecture* "describes the static organization of software into subsystems interconnected through interfaces and defines at a significant level how these software subsystems interact with each other." (Garlan and Shaw, 1996); others include some of the essential behavior and lkey mechanisms in the definition of architecture as well, using use-cases and interaction diagrams to capture these.Similar to how a building architect works with customers and suppliers to analyze requirements and technology trends to design a building, a software *architect* “defines and maintains the architecture of a system, that is, the essential part of the use case, design, implementation and test models; the architect decides on which architectural styles and patterns to use in the system.” (Jacobson, 1994). I believe this distinct role of architect and architecture is especially important for families of large-scale systems and product-lines.

Components and component systems

In my book I use the term *component* for any reusable element of a development model that is loosely coupled to other elements and is designed and packaged for reuse. Since this expands on the definition in Chapter 1 and 3 and might be confusing with other work in this book, we will to use a slightly modified terminology that distinguishes reusable components from other reusable elements that are part of the full specification and elaboration of a component - these we will call component elements. However, we can not invent totally new terminology, in order to keep some alignment with the RSEB book. Any model element or software engineering work product can be designed for reuse and reused when a new work product is developed. Work products intended for systematic reuse must be designed, packaged and documented for reuse. Candidates include: use cases, classes, interfaces, patterns, tests, and source code (such as Java Beans, Ada packages, C++ code, VB script). In the RSEB book, we called all of these components. These are more than just modules, because are they are designed to conform to an architecture, and complement or complete a component.

A *component system* is a group of reusable components and component elements connected by relationships and interfaces that interact with each other within a layered, component infrastructure. A component system exposes to potential reusers though a *façade* the minimal information and model elements needed to effectively reuse the component system. (Note that façade is a UML 1.3 and RSEB term describing a package of public elements, available for export; it is not the same as an interface, and may include any model elements intended to be reused as the models for an application are built. The facade essentially behaves as a subset model describing a consistent external view of the component system. In some ways, it extended and enriches the more familiar notion of interface).

The commonality and variability in a product-line is made explicit through *variation points* and *variants* in the components and other reusable component elements. Some reusable work products can be used “as is”; others must be specialized before use. A variety of mechanisms can be used to implement variability (such as parameters, inheritance, extensions, templates, or generators), as described below.

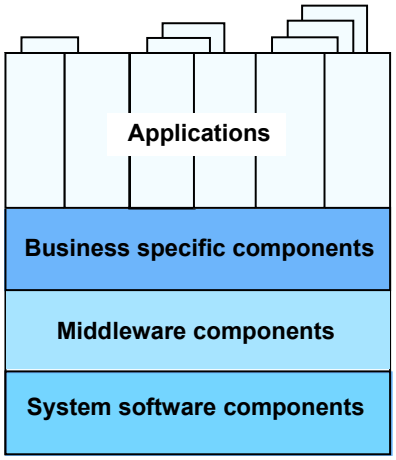

Figure 3: The RSEB component infrastructure contains three layers of component types

Layered Component Infrastructure

It is difficult to create a coherent set of compatible, reusable, maintainable parts through *ad hoc* reuse of existing software, or even by the use of common platforms and libraries. You can only create reusable parts by developing and maintaining a reuse infrastructure (architecture, frameworks, and components) as an organizational asset.

We construct applications as layered systems with a modular, layered architecture. Each application is represented as a separate Application System (a consistent set of models and other work products), built using lower-level components and component elements drawn from lower-level component systems. Layers below the application layer contain components targeted for specific business or application domain areas (such as banking systems or microwave instruments), common middleware cross-business components (including object request brokers (ORBs), databases, and graphical user interfaces (GUIs)), and platform-specific (hardware and software) software components and interfaces (such as operating system, networking, and specialized devices).

This layered architecture and component infrastructure provides components and a roadmap to help each person in the organization understand and apply the desired engineering practice. The component infrastructure supports platform-independent *interfaces* providing openness and flexibility. Different implementations of the same interfaces can be plug-compatible. Typically, provided and needed interfaces will be packaged with other related reusable model elements in one or more facades. Finally, the infrastructure allows component systems to evolve independently, as new technologies and opportunities arise.

Applications and application systems

Applications are defined by a set of connected UML models and other work products, called an *Application System.* Application engineers (architects, designers, and implementers) and other "reusers" (such as component engineers) construct software applications by selecting components and integrating them together to form a complete system. At an early stage of software development, they would work with reusable modular use cases. At later stages, they work with reusable design components or code classes. It is important for developers to try to integrate the use cases from multiple components, because in doing so they will uncover problems that would have occurred when the respective components were integrated.

Product-Line CBSE Process and Organization

A product-line CBSE creates component systems by one or more teams for reuse by other teams. Instead of building each related application independently, the organization purposefully and proactively creates reusable assets that are then used to build applications more rapidly and cost effectively. The RSEB links previously independent projects, introduces new component infrastructures, changes development processes, introduces reusable component management and funding activities, and changes the roles of

designers, process engineers, development engineers, and managers. Significant organizational change is involved.

RSEB applies business process engineering to the software engineering organization itself. Business use cases model the core software development processes, such as those followed by component engineers and application engineers. These models are further refined to define the roles and responsibilities of the workers, the workflows they enact, and the information systems and tools they use. Different structures of the business models allow us to model various organizations.

Shown as a business use case in Figure 2, the three main categories of software engineering processes within an RSEB are: *Application Family Engineering*, *Application System Engineering*, and *Component System Engineering*.

Application family engineering

The *application family engineering* process creates the layered system architecture and component infrastructures for the product-line and determines how to decompose the overall set of applications into application systems and supporting component systems. This architectural process is a design endeavor that creates the layers, façades, and interfaces of the subsystems and component systems that support the complete product-line. The application family engineer (an architect or senior designer) must:

1. Understand the requirements that current users of similar systems know they need now, and in addition what potential users think they might want to have in future applications;
2. Develop a layered infrastructure robust enough to survive the inevitable changes that will occur;
3. Identify component systems as well as individual applications; and
4. Wrap, re-engineer, or interface with existing software, such as legacy systems and/or component producer systems.

Application system engineering

The *application system engineering* process selects, specializes, and assembles components and component elements from one or more component systems (made viisble through an appropriate façade) into complete application systems. It uses appropriate tools, methods, processes, and instructions provided explicitly with the component system. The process begins when a customer requests a new version of an application. Developers (sometimes including architects) first elicit the requirements from a few sources, primarily the customers and the end-users. Then the developers express the requirements in terms of available component infrastructures. Certain requirements are met by directly reusing some components or specializing others. If the overall infrastructure is well designed, and if a comprehensive set of components are available, developers (aided by librarians) can find an appropriate component or existing component element to reuse. When no appropriate reusable component or other workproduct is available, the developers may have to design a new component and software to meet the requirement. A reuser may exploit variability mechanisms to adapt components to the particular application. During the modeling and implementation of an application or component system, the reuser can insert pre-supplied or custom-built variants (which could be a complete subsidiary component or some other reusable component element, or some other reuser developed software element) at designated variation points to produce a specialized element. For example, if the variability mechanism is explicit parameters for a blackbox component, then setting the parameters to specific values, including perhaps binding some references to other components, is the "attaching." Instead, if the component define explicit required interfaces, smaller components or other computational elements that match these interfaces must be "plugged in." Finally, if the component is a framework that defines abstract or concrete classes, then concrete subclasses must be supplied that inherit from these base classes to complete the application. Thus, an application system is both customizable and configurable.

Component system engineering

The *component system engineering* process designs, constructs, and packages components into component systems. The process uses appropriate code, templates, models, dictionaries, documents, and perhaps custom tools. The process begins when a reuse business identifies a new component system to design. The architects and designers must elicit and analyze requirements about current needs and future trends from

multiple sources, including: business models, architects, domain experts, and application users. The goal is to create a set of reusable components that expresses commonality and variability appropriate to the family of applications. Architects and developers (depending on the scope of the decision)should calculate costs and benefits for the amount of functionality and variability to incorporate in the components. Many components will be designed with variation points and pre-built variants to increase their intended use. In addition to using inheritance as a well-known (and unfortunately, sometimes overused) specialization mechanism, you can also use problem-oriented languages, aspects, parameterized templates and generators (Bassett 1996; Griss 2000).

The process concludes with the certification (including internal testing, final inspection and acceptance testing) and packaging (including documentation, classification, examples and comparisons with other components) of the component system for retrieval by potential reusers. See also Hedley Apperly's chapter XX.

Incremental Transition to an RSEB

The RSEB uses a business process reengineering approach developed to systematically transition an existing software organization into an RSEB (Jacobson, 1994). Each step in the process includes specific organizational change management guidelines, such as the use of champions, and some reuse pragmatics, such as the use of incremental, pilot-driven reuse adoption, and distinct reuse-maturity stages. To develop the appropriate transition schedule, an organization must assess important business and domain drivers, as well as its organization and process maturity.

Reuse and Process Maturity

A key question that arises when considering the transition to systematic CBSE is that of organizational process maturity, and specifically, the implications of software engineering process maturity, as expressed by the Software Engineering Institute's Capability Maturity Model (CMM) (Paulk 1993; Humphrey 1996). Indeed, reuse and the CMM are strongly related because introducing systematic software reuse is a significant process improvement, driven by critical business need. The CMM is a process improvement framework that summarizes key guidelines about mature software process that can be directly applied to improved reuse practice (Humphrey 1996; Paulk, 1993). The change to a set of concurrent, managed and supported multi-project processes demands significant organization changes and standardized process throughout the organization. In general, it makes good business sense for an effective plan to address the incremental reuse adoption program together with the SEI process improvement program (Griss 1998a, 1998b). Many software engineers and managers who know the CMM believe that they should delay reuse until CMM Level 3 (*Defined Process*) is achieved, that is, when the entire organization is able to follow an explicitly defined process. However, although cost-effective, organization-wide reuse requires the discipline and formal processes characteristic of CMM Level 3 or higher, significant progress to increased reuse can be made with lower CMM levels. Even a lesser amount of reuse can be of significant value in reaching critical business goals such as reduced time to market. For more details, see (Griss 1998a, 1998b).

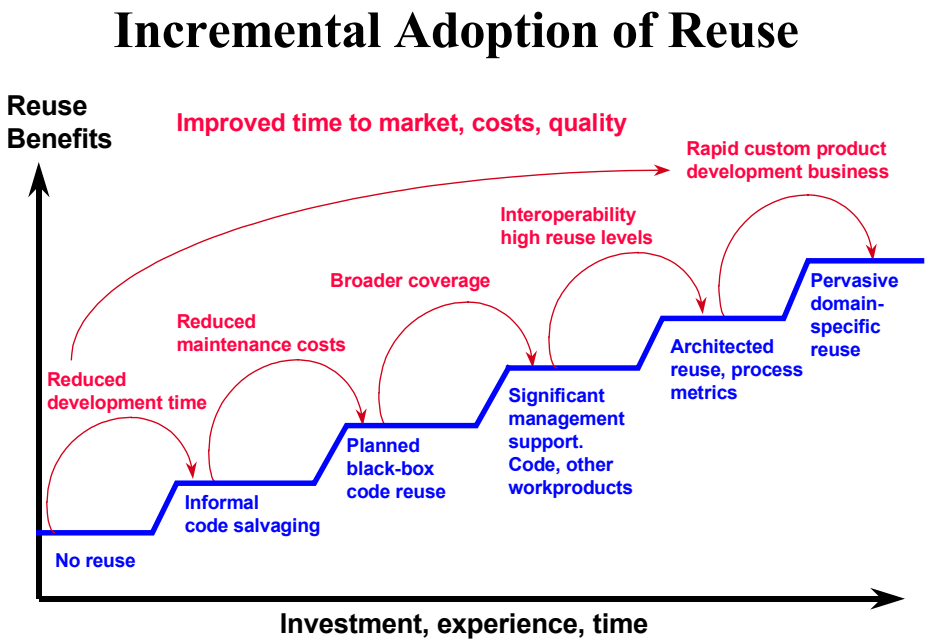

Figure 1: Incremental adoption of reuse is driven by compelling business need.

Incremental Adoption of CBSE

I have found it is best to introduce CBSE incrementally, making organization and process changes in a series of steps. For most organizations, it is best to focus staged improvement steps on a set of pilot projects. The stages are summarized by a simplified Reuse Maturity Model (RMM), shown in Figure 3, based in part on experience gained from work at HP and the use of the CMM. An organization becomes ready for a transition from one stage to another when its business needs are compelling enough to motivate change.

These stages are:

RMM 1 - No reuse: Some code sharing may occur, but people work independently on unrelated projects. They do not communicate about their code and often take pride in "doing it all" themselves.

RMM 2 - Informal code salvaging: When developers trust each other and their code, they begin to copy and adapt chunks of code from one system into new systems. Although they might prefer to rewrite the software, they copy the code to reduce time spent on a project.

RMM 3 - Planned black-box code reuse: While informal code salvaging reduces development time and testing, maintenance problems soon increase. Multiple copies of components, each slightly different, have to be managed. Defects found in one copy have to be found and fixed multiple times. The next stage is a planned "black box" code reuse strategy in which carefully chosen instances of code are reengineered into components, tested and documented for reuse, and reused without change.

RMM 4 - Managed work product reuse: To increase organization-wide reuse, you need a process that supports an increasing number of reusers. Should everyone be “forced” to use only the standard version? Should multiple versions be maintained? Should adaptation be allowed? Who decides? What else should be reused? This stage leads to a defined component management process, with distinct creator and reuse projects, and a support organization. Employees need education and help in using these components. Strong configuration management processes and tools are needed.

RMM 5 - Architected reuse: To achieve higher levels of reuse, and gain increased coverage from design to implementation, it is important to design the components and the infrastructure that will use them.

Developing and adhering to common architectures and component infrastructures involves even more organizational commitment and structure than ad hoc reuse of a unrelated components. Groups of developers have to work together to agree on, and then enforce, interfaces and feature sets. Modeling notations become critical.

RMM 6 - Pervasive domain-specific product-line CBSE: Product-lines are planned, and component infrastructures and components are defined to ensure maximum reuse. Component development is carefully scheduled and resourced to ensure the quickest return on the reuse investment. People specialize in different roles, such as design for reuse, domain engineering, component engineering and reuse library management. Separate teams operate in a concurrent, coordinated way.

Benefits, such as improved time to market, higher-quality systems, or lower overall development costs, increase as the levels of reuse and the sophistication of the reuse program increase. Organizations cannot easily jump steps needed to achieve mastery at a particular level, although they can begin to master and institutionalize skills on one level while exploring the higher levels.

Conclusion

Effective systematic reuse is directly related to increased organizational process maturity. An organization can evolve a product-line CBSE reuse business by becoming more skilled and disciplined in following standard processes, as well as using and creating standard templates, documents and software work products. Effective product-line CBSE requires a coherent approach that: designs a product-line component infrastructure and components, organizes the workforce, and enables the transition of the development and business processes. I recommended the following key steps to ensure your own “CBSE success story”:

1. Clarify the business goals that motivate product-line CBSE. Customize the CBSE program to meet these goals. Assure that senior management supports CBSE and reuse throughout the organization.
2. Address a significant segment of the product-line or key domain with enough projected reuse of the components to justify the extra technical and management effort. Ensure that senior management and the engineering staff understand that the payoff will occur only if there is repeated use.
3. Assess process maturity, experience and readiness of the organization, and plan an incremental adoption roadmap to move the organization to mature CBSE. Use fast-paced pilot projects and reengineer existing software into initial components, as appropriate. This will test and demonstrate the CBSE program for an important subset of the desired architecture, components, and product-line.
4. Design and incrementally implement a layered architecture and infrastructure for the product family. Develop components and component systems, exploiting variability mechanisms appropriate to the domain or product-line variability and to the developer process and tool maturity.
5. Match the organization to the product-line structure. Assign key component and application systems to distinct parts of the organization.

References

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

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

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

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

J Favaro, K Favaro and P Favaro "Value Based Software Reuse Investment," *Annals of Software Engineering (5),* 1998.

WB Frakes and S Isoda. "Success factors of systematic reuse." *IEEE Software, 11(5)*: 15-19, Sep 1994.

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

ML Griss "Software Reuse: A process of getting organized." *Object Magazine,* May 1995.

ML Griss, “Domain Engineering And Variability In The Reuse-Driven Software Engineering Business,” *Object Magazine*, Dec 1996.

ML Griss, “Improved Cycle Time in the Reuse-driven Software Engineering Business,” *Object Magazine,* Aug 1997.

ML Griss, "Reuse Strategies - Models and Patterns of Success." *Component Strategies*. SIGS. Oct. 1998.

ML Griss, “CMM as a Framework for Systematic Reuse - part 1,” *Object Magazine*. Mar 1998(a).

ML Griss, "CMM, PSP and TSP as a Framework for Adopting Systematic Reuse - part 2," *Object Magazine*. June 1998 (b).

ML Griss, "Implementing Product-Line Features with Component Reuse,” *Proceedings of the 6th International Conference on Software Reuse*, Austria, Springer-Verlag, June 2000 (To appear).

WS Humphrey, “Using a Defined and Measured Personal Software Process,” *IEEE Software,* May 1996, pp. 77-88. (See also http://www.sei.cmu.edu/)

I Jacobson et al, *Object-oriented software engineering: A use case driven approach*, Addison-Wesley, 1992.

I Jacobson, M Ericsson and A Jacobson*, The Object Advantage: Business process reengineering with object technology,* Addison-Wesley 1994.

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

Philippe Kruchten, The 4+1 view model of architecture, *IEEE Software, Nov 1995, V12, #6, pp. 42-50.*

T. Mowbray and R Malveau CORBA Design Patterns. John Wiley & Sons, 1997

MC Paulk et al, “Capability Maturity Model, Version 1.1”, *IEEE Software 10(4),* July 1993, pp.18-27. (See also http://www.sei.cmu.edu)

G Pour, "Component-Based Software Development: New Opportunities and Challenges," *Proceedings of the 26th International Conference on Technology of Object-Oriented Systems and Languages (TOOLS USA’‘98),* Santa Barbara, CA, August 1998.

Terminology Sidebar

- A *product-line* is a set of products that share a common set of requirements, but also exhibit significant variability in requirements.
- A *feature* is a product characteristic that users and customers view as important in describing and distinguishing members of the product-line. *Feature-driven* development has each release defined by the features in included or omits, and focuses has teams of people focused on developing software for sets of related) features.
- A *framework* is a skeletal product, comprising an implementation of key components, component elements, infrastructure and mechanisms that are common to all members of the product-line.
- *Modularity* refers to the decomposition and packaging of pieces of software so as to hide decisions and details, and thus decrease coupling between parts of the system. An extreme form of modularity is the component, that completely packages its details behind well defined interfaces, has a well-defined deployment lifecycle, hiding some execution details, and conforms to a well-defined infrastructure.

- An *interface* is a set of method signatures that a particular component promises to implement, or requires that other components provide. The same method signature may be mentioned in multiple interfaces provided or required by a component.
- A *façade* is a packaging of a set of component elements and other workproducts, exported from a component system, intended to be imported and reused to create the model elements and workproducts in some other system. A component system can offer multiple façades, each collecting a set of related elements for convenient and consistent reuse. Some of the elements in a façade may include interfaces, but other elements such as usecases, variants, and designs can also be present.