A Framework for Software Product Line Practice Version 4.2

Introduction

A product line is a set of products that together address a particular market segment or fulfill a particular mission. Product lines are, of course, nothing new in manufacturing. Boeing builds one, and so do Ford, Dell, and even McDonald's. Each of these companies exploits commonality in different ways. Boeing, for example, developed the 757 and 767 transports in tandem, and the parts lists for these very two different aircraft overlap by about 60%, achieving significant economies of production and maintenance. But *software* product lines based on inter-product commonality are a relatively new concept that is rapidly emerging as a viable and important software development paradigm. Product flexibility is the anthem of the software marketplace, and product lines fulfill the promise of tailor-made systems built specifically for the needs of particular customers or customer groups. A product line succeeds because the commonalities shared by the software products can be exploited to achieve economies of production. The products are built from common assets in a prescribed way.

Companies are finding that this practice of building sets of related systems from common assets can yield remarkable quantitative improvements in productivity, time to market, product quality, and customer satisfaction. They are finding that a software product line can efficiently satisfy the current hunger for mass customization. Organizations that acquire, as opposed to build, software systems are finding that commissioning a set of related systems as a commonly developed product line yields economies in delivery time, cost, simplified training, and streamlined acquisition.

But along with the gains come risks. Using a product line approach constitutes a new technical strategy for the organization. Organizational and management issues constitute obstacles that are at least as critical to overcome and often add more risk because they are less obvious. Building a software product line and bringing it to market requires a blend of skillful engineering as well as both technical and organizational management. Acquiring a software product line also requires this same blend of skills to position the using organizations to effectively exploit the commonality of the incoming products, as well as to lend sound technical oversight and monitoring to the development effort. These skills are necessary to overcome the pitfalls that may bring failure to an unsophisticated organization.

We've worked to gather information and identify key people with product line experience. Through surveys, workshops, conferences, case studies, and direct collaboration with organizations on product line efforts, we have amassed and categorized a reservoir of information. Organizations that have succeeded with product lines vary widely in

- the nature of their products
- their market or mission
- their business goals
- their organizational structure
- their culture and policies
- their software process discipline
- the maturity and extent of their legacy artifacts

Nevertheless, there are universal essential activities and practices that emerge, having to do with the ability to construct new products from a set of common assets while working under the constraints of various organizational contexts and starting points. This document describes a framework[1] for product line development. The framework is an on-line product line encyclopedia; it is a web-based document describing the essential activities and practices, in both the technical and organizational areas. These activities and practices are those in which an organization must be competent before it can reap the maximum benefit from fielding a product line of software or software-intensive systems. The audience for this framework includes members of an organization who are in a position to make or influence decisions regarding the adoption of product line practices as well as those who are already involved in a product line effort.

Purpose

The goals of this framework are

- **to identify the foundational concepts underlying software product lines and the essential activities to consider before developing a product line**
- **to identify practice areas that an organization developing software product lines must master**
  Although these practice areas may be required for engineering any software system, the product line context imposes special constraints so that they must be carried out in a non-conventional way.
- **to define practices in each practice area, where current knowledge is sufficient to do so**
  For example, "Configuration Management" is a practice area that applies to any software development effort, but it has special implications for product line development. Thus, we identify "Configuration Management" as a practice area, but we also are able to define one or more effective configuration management practices for product lines. In many cases, the definition of the practice is a reference to a source outside this document.
- **to provide guidance to an organization about how to move to a product line approach for software**

An organization using this framework should be able to understand the state of its product line capabilities by (a) understanding the set of essential practice areas, (b) assessing how practices in those areas differ from their conventional forms for single product development, and (c) comparing that set of practices to the organization's existing skill set.

As such, this framework can serve as the basis for a technology and improvement plan aimed at achieving product line development goals.

Every organization is different and comes to the product line approach with different goals, missions, assets, and requirements. Practices for a product line builder will be different from those for a product line acquirer, and different still for a component vendor. Appropriate practices will also vary according to

- the type of system being built
- the depth of domain experience
- the legacy assets on hand
- the organizational goals
- the maturity of artifacts and processes
- the skill level of the personnel available
- the production strategy embraced
- many other factors

There is no one correct set of practices for every organization; we do not prescribe a methodology consisting of a set of specific practices. The framework is not a maturity model[1] or a process guide. We are prescriptive about the practice areas and we do prescribe that organizations adopt appropriate practices in each practice area. This document contains practices that we have seen work successfully.

The framework has been used by organizations, large and small, to help them plan for their adoption of the product line approach, as well as to help them gauge how they're doing and in what areas they're falling short. We use it to guide our collaborations with customers and to focus in what areas our collaboration will best assist our customers. We also use it as the basis for conducting Product Line Technical Probes, which are formal diagnostics of an organization's product line fitness [Clements 01a, Chapter 8; see also http://www.sei.cmu.edu/productlines/pltp.html]. The framework is a living, growing document; it represents our best picture of sound product line practice as described to us by its many reviewers and users: all of whom are practitioners. The framework is specifically about software product lines and as such has served successfully as the basis for technology and improvement plans aimed at achieving product line goals. We understand that since its first release organizations have found the framework very useful in product lines not related to software. We make no claims about its utility in non-software contexts but recognize that many of the underlying principles and practices are likely relevant.

What is a Software Product Line?

A *software product line* is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

This definition is consistent with the definition traditionally given for any product line. But it adds more; it puts constraints on the way in which the systems in a software product line are developed. Why? Because substantial production economies can be achieved when the systems in a software product line are developed from a common set of assets in a prescribed way, in contrast to being developed separately, from scratch or in an arbitrary fashion. It is exactly these production economies that make the software product line approach attractive.

How is production made more economical? Each product is formed by taking applicable components from the base of common assets, tailoring them as necessary through preplanned variation mechanisms such as parameterization or inheritance, adding any new components that may be necessary, and assembling the collection according to the rules of a common, product-line-wide architecture. Building a new product (system) becomes more a matter of assembly or generation than one of creation; the predominant activity is integration rather than programming. For each software product line there is a predefined guide or plan that specifies the exact product-building approach.

Certainly the desire for production economies is not a new business goal, and neither is a product line solution. But a *software product line* is a relatively new idea, and it should seem clear from our description that software product lines require a different technical tack. The more subtle consequence is that software product lines require much more than new technical practices.

The common set of assets and the plan for how they are used to build products don't just materialize without planning, and they certainly don't come free. They require organizational foresight, investment, planning, and direction. They require strategic thinking that looks beyond a single product. The disciplined use of the common assets to build products doesn't just happen either. Management must direct, track, and enforce the use of the assets. Software product lines are as much about business practices as they are about technical practices.

Software product lines give *economies of scope*, which means that you take economic advantage of the fact that many of your products are very similar—not by accident, but because you planned it that way. You make deliberate, strategic decisions and are systematic in effecting those decisions.

What Software Product Lines Are Not

There are many approaches that at first blush could be confused with software product lines. Describing what you don't mean is often as instructive as describing what you do mean. When we speak of software product lines, we don't mean any of the following:

Fortuitous Small-Grained Reuse

Reuse, as a software strategy for decreasing development costs and improving quality, is not a new idea, and software product lines definitely involve reuse—reuse, in fact, of the highest order. So what's the difference? Past reuse agendas have focused on the reuse of relatively small pieces of code—that is, small-grained reuse. Organizations have built reuse libraries containing algorithms, modules, objects, or components. Almost anything a software developer writes goes into the library. Other developers are then urged (and sometimes required) to use what the library provides instead of creating their own versions. Unfortunately, it often takes longer to locate these small pieces and integrate them into a system than it would take to build them anew. Documentation, if it exists at all, might explain the situation for which the piece was created but not how it can be generalized or adapted to other situations. The benefits of small-grained reuse depend on the predisposition of the software engineer to use what is in the library, the suitability of what is in the library for the engineer's particular needs, and the successful adaptation and integration of the library units into the rest of the system. If reuse occurs at all under these conditions, it is fortuitous and the payoff is usually nonexistent.

In a software product line approach, the reuse is planned, enabled, and enforced—the opposite of opportunistic. The asset base includes those artifacts in software development that are most costly to develop from scratch—namely, the requirements, domain models, software architecture, performance models, test cases, and components. All of the assets are designed to be reused and are optimized for use in more than a single system. The reuse with software product lines is comprehensive, planned, and profitable.

Single-System Development with Reuse

Suppose you are developing a new system that seems very similar to one you have built before. You borrow what you can from your previous effort, modify it as necessary, add whatever it takes, and field the product, which then assumes its own maintenance trajectory separate from the first. What you have done is what is called "clone and own." You certainly have taken economic advantage of previous work; you have reused a part of another system. But now you have two entirely different systems, not two systems built from the same base. This is again ad hoc reuse.

There are two major differences between this approach and a software product line approach. First, software product lines reuse assets that were designed explicitly for reuse. Second, the product line is treated as a whole not as multiple products that are viewed and maintained separately. In mature product line organizations, the concept of multiple products disappears. Each product is simply a tailoring of the common assets, which constitute the core of each product, plus perhaps a small collection of additional artifacts unique to that product. It is the core assets that are designed carefully and evolved over time. It is the core assets that are the organization's premiere intellectual property.

Just Component-Based Development

Software product lines rely on a form of component-based development, but much more is involved. The typical definition of component-based development involves the selection of components from an in-house library or the marketplace to build products. Although the products in software product lines certainly are composed of components, these components are all specified by the product line architecture. Moreover, the components are assembled in a prescribed way, which includes exercising built-in variability mechanisms in the components to put them to use in specific products. The prescription comes from both the architecture and the production plan, and is missing from standard component-based development. In a product line, the generic form of the component is evolved and maintained in the core asset base. In component-based development, if any variation is involved, it is usually accomplished by writing code, and the variants are most likely maintained separately. Component-based development also lacks the technical and organizational management aspects that are so important to the success of a software product.

Just a Reconfigurable Architecture

Reference architectures and object-oriented frameworks are designed to be reused in multiple systems and to be reconfigured as necessary. Reusing architectural structures is a good idea because the architecture is a pivotal part of any system and a costly one to construct. A product line architecture is designed to support the variation needed by the products in the product line, and so making it reconfigurable makes sense. But the product line architecture is just one asset, albeit an important one, in the product line's core asset base.

Releases and Versions of Single Products

Organizations routinely produce new releases and versions of products. Each of these new versions and releases is typically constructed using the architecture, components, test plans, and other features of the prior releases. Why are software product lines different? First, in a product line there are multiple simultaneous products, all of which are going through their own cycles of release and versioning simultaneously. Thus, the evolution of a single product must be considered within a broader context—namely, the evolution of the product line as a whole. Second, in a single-product context, once a product is updated there's often no looking back—whatever went into the production of earlier products is no longer considered to be of any value, or at best, retired as soon as practicable. But in a product line, an early version of a product that is still considered to have market potential can easily be kept as a viable member of the family: it is, after all, an instantiation of the core assets, just like other versions of other products.

Just a Set of Technical Standards

Many organizations set up technical standards to limit the choices their software engineers can make regarding the kinds and sources of components to incorporate in systems. They audit for compliance at architecture and design reviews to ensure that the standards are being followed. For example, the developer might be able to select between three identified database choices and two identified Web browsers, but must use a specific middleware or spreadsheet product if either is necessary. Technical standards are constraints to promote interoperability and to decrease the cost associated with maintenance and support of commercial components. An organization that undertakes a product line effort may have such technical standards, in which case the product line architecture and components will need to conform to those standards. However, the standards are simply constraints that are input to the software product line, no more.

Benefits and Costs of a Product Line

Software product line approaches accrue benefits at multiple levels. This section lists the benefits (and some of the costs) from the perspective of the organization as a whole, individuals within the organization, and the core assets involved in software product line production.

Organizational Benefits

The organizations that we have studied[1] have achieved remarkable benefits that are aligned with commonly held business goals. Some of these include:

- large-scale productivity gains
- decreased time-to-market
- increased product quality
- increased customer satisfaction
- more efficient use of human resources
- ability to effect mass customization
- ability to maintain market presence
- ability to sustain unprecedented growth

These benefits give organizations a competitive advantage. They are derived from the reuse of the core assets in a strategic and prescribed way. Once the product line core asset repository is established, there is a direct savings *each* time a product is built, associated with *each* of the following:

- **Requirements:** There are common product line requirements. Product requirements are deltas to this established requirements base. Extensive requirements analysis is saved. Feasibility is assured.
- **Architecture:** An architecture for a software system represents a large investment of time from the organization's most talented engineers. The quality goals for a system—its performance, reliability, modifiability, and so on—are largely allowed or precluded once the architecture is in place. If the architecture is wrong, the system cannot be saved. The product line architecture is used for each product and need only be instantiated. Considerable time and risk are spared.

- **Components:** Up to 100% of the components in the core asset base are used in each product. These components may need to be altered using inheritance or parameters, but the design is intact, as are data structures and algorithms. In addition, the product line architecture provides component specifications for all but any unique components that may be necessary.
- **Modeling and analysis:** Performance models and the associated analyses are existing product line core assets. With each new product there is extremely high confidence that the timing problems have been worked out and that the bugs associated with distributed computing—synchronization, network loading, and absence of deadlock—have been eliminated.
- **Testing:** Generic test plans, test processes, test cases, test data, test harnesses, and the communication paths required to report and fix problems have already been built. They need only be tailored on the basis of the variations related to the product.
- **Planning:** The production plan has already been established. Baseline budgets and schedules from previous product development projects already exist and provide a reliable basis for the product work plans.
- **Processes:** Configuration control boards, configuration management tools and procedures, management processes, and the overall software development process are in place, have been used before, and are robust, reliable, and responsive to the organization's special needs.
- **People:** Fewer people are required to build products and the people are more easily transferred across the entire line.

Product lines enhance quality. Each new system takes advantage of all of the defect elimination in its forebears; developer and customer confidence both rise with each new instantiation. The more complicated the system, the higher the payoff for solving the vexing performance, distribution, reliability, and other engineering issues once for the entire family.

Individual Benefits

The benefits to individuals within an organization depend upon their respective roles. The following table shows observed benefits for some of the individual stakeholders in the product line organization.

Product Line Benefits for Individual Stakeholders

Stakeholder Role	Benefits
CEO	Large productivity gains; greatly improved time to market; sustained growth and market presence; ability to economically capture a market niche.
COO	Efficient use of work force; ability to explore new markets, new technology, and/or new products; fluid personnel pool.
Technical Manager	Increased predictability; well-established roles and responsibilities; efficient production.
Software Product	Higher morale; greater job satisfaction; can focus on truly unique aspects of

Developer	products; easier software integration; fewer schedule delays; greater mobility within the organization; more marketable; have time to learn new technology; are part of a team building products with an established quality record and reputation.
Architect or Core Asset Developer	Greater challenge; work has more impact; prestige within the organization; as marketable as the product line.
Marketer	Predictable high quality products; predictable delivery; can sell products with a pedigree.
Customer	Higher quality products; predictable delivery date; predictable cost; known costs for unique requirements; well-tested training materials and documentation; shared maintenance costs; potential to participate in a user's group.
End User	Fewer defects; better training materials and documentation; a network of other users.

Benefits versus Costs

We have established that the strategic reuse of core assets that defines product line practice represents an opportunity for benefits across the board, but the picture is not yet complete. Launching a software product line is a business decision that should not be made randomly. Any organization that launches a product line should have in mind specific and solid business goals that it plans to achieve through product line practice. Moreover, the benefits given above should align carefully with the achievement of those goals, because a software product line requires a substantial start-up investment as well as ongoing costs to maintain the core assets. We have already listed the benefits associated with the reuse of particular core assets. Usually a cost and a caveat are associated with the achievement of each benefit. The following table gives a partial list of core assets with the typical additional costs. We repeat the benefits for the sake of comparison.

Costs and Benefits of Product Lines		
Core Asset	**Benefit**	**Additional Cost**
Requirements: The requirements are written for the group of systems as a whole, with requirements for individual systems specified by a delta or an increment to the generic set.	Commonality and variation are documented explicitly, which will help lead to an architecture for the product line. New systems in the product line will be much simpler to specify because the requirements are reused and tailored.	Capturing requirements for a group of systems may require sophisticated analysis and intense negotiation to agree on both common requirements and variation points acceptable for all the systems.
Architecture: The architecture for the product line is the blueprint for how each product is assembled from the components in the core asset base.	Architecture represents a significant investment by the organization's most talented engineers. Leveraging this investment across all products in the product line means that for subsequent products, the most important design step is largely completed.	The architecture must support the variation inherent in the product line, which imposes an additional constraint on the architecture and requires greater talent to define.

Software components: The software components that populate the core asset base form the building blocks for each product in the product line. Some will be reused without alteration. Others will be tailored according to prespecified variation mechanisms.	The interfaces for components are reused. For actual components that are reused, the design decisions, data structures, algorithms, documentation, reviews, code, and debugging effort can all be leveraged across multiple products in the product line.	The components must be designed to be robust and extensible so that they are applicable across a range of product contexts. Variation points must be built in or at least anticipated. Often, components must be designed to be more general without loss of performance.
Performance modeling and analysis: For products that must meet real-time constraints (and some that have soft real-time constraints), analysis must be performed to show that the system's performance will be adequate.	A new product can be fielded with high confidence that real-time and distributed-systems problems have already been worked out, because the analysis and modeling can be reused from product to product. Process scheduling, network traffic loads, deadlock elimination, data consistency problems, and the like will all have been modeled and analyzed.	Reusing the analysis may impose constraints on moving of processes among processors, on creation of new processes, or on synchronization of existing processes.
Business case, market analysis, marketing collateral, cost and schedule estimates: These are the up-front business necessities involved in any product. Generic versions are built that support the entire product line.	All of the business and management artifacts involved in turning out already exist at least in a generic form and can be reused.	All of these artifacts must be generic, or be made extensible to accommodate product variations.
Tools and processes for software development and for making changes: The infrastructure for turning out a software product requires specific product line processes and appropriate tool support.	Configuration control boards, configuration management tools and procedures, management processes, and the overall software development process are in place and have been used before. Tools and environments purchased for one product can be amortized across the entire product line.	The boards, process definitions, tools, and procedures must be more robust to account for unique product line needs and for the differences between managing a product line and managing a single product.
Test cases, test plans, test data: There are generic testing artifacts for the entire set of products in the product line with variation points to accommodate product variation.	Test plans, test cases, test scripts, and test data have already been developed and reviewed for the components that are reused. Testing artifacts represent a substantial organizational investment. Any saving in this area is a benefit.	All of the testing artifacts must be more robust because they will support more than one product. They also must be extensible to accommodate variation among the products.
People, skills, training: In a product line organization, even though members of the	Because of the commonality of the products and the production process, personnel can be more	Personnel must be trained beyond general software engineering and corporate

development staff may work on a single product at a time, they are in reality working on the entire product line. The product line is a single entity that embraces multiple products.	easily transferred among product projects as required. Their expertise is usually applicable across the entire product line. Their productivity, when measured by the number of products to which their work applies, rises dramatically. Resources spent on training developers to use processes, tools, and system components are expended only once.	procedures to ensure that they understand software product line practices and can use the core assets and procedures associated with the product line. New personnel must be much more specifically trained for the product line. Training materials must be created that address the product line. As product lines mature, the skills required in an organization tend to change, away from programming and toward relevant domain expertise and technology forecasting. This transition must be managed.

For each of these core assets, the investment cost is usually much less than the value of the benefit. Also, most of the costs are up-front costs associated with establishing the product line. The benefits, on the other hand, accrue with each new product release. Once the approach is established, the organization's productivity accelerates rapidly and the benefits far outweigh the costs. However, an organization that attempts to institute a product line without being aware of the costs is likely to abandon the product line concept before seeing it through.

It takes a certain degree of maturity in the developing organization to field a product line successfully. Technology change is not the only barrier to successful product line adoption. Changes in management and organizational practices are also involved. Successful adoption of software product line practice is a careful blend of technological, process, organizational, and business improvements.

Organizations of all stripes have enjoyed quantitative benefits from their product lines. Product line practitioners have also shared with us examples of the costs, such as:

- canceling three large projects so that sufficient resources could be devoted to the up-front development of core assets
- reassigning staff who could not adjust to the product line way of doing business
- suspending product delivery for an extended period while putting the new practices into place

Certainly not every organization must undergo such dramatic measures in order to adopt the software product line approach. And the companies that bore these costs and made the successful transition to product line practice all agree that the payoff more than compensated for the effort. But these costs underscore the point that product line practice is often uncharted territory and may not be the right path for every organization.

1. The SEI has published several detailed case studies of successful product line organizations and the benefits they have enjoyed. These may be found in [Clements 01a] as well as http://www.sei.cmu.edu/productlines/plp_publications.html

A Note on Terminology

In this document we use the following terms:

A *software product line* is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. This is the definition we provided in What is a Software Product Line.

Core assets are those reusable artifacts and resources that form the basis for the software product line. Core assets often include, but are not limited to, the architecture, reusable software components, domain models, requirements statements, documentation and specifications, performance models, schedules, budgets, test plans, test cases, work plans, and process descriptions. The architecture is key among the collection of core assets.

Development is a generic term used to describe how core assets (or products) come to fruition. Software enters an organization in any one of three ways: the organization can build it itself (either from scratch or by mining legacy software), purchase it (buy it, largely unchanged, off the shelf), or commission it (contract with someone else to develop it especially for them). So our use of the term "development" may actually involve building, acquisition, purchase, retrofitting earlier work, or any combination of these options. We recognize and address these options, but we use "development" as the general term.

A *domain* is a specialized body of knowledge, an area of expertise, or a collection of related functionality. For example, the telecommunications domain is a set of telecommunications functionality, which in turn consists of other domains such as switching, protocols, telephony, and network. A telecommunications software product line is a specific set of software systems that provide some of that functionality.

Software product line practice is the systematic use of core assets to assemble, instantiate, or generate the multiple products that constitute a software product line. The choice of verb depends on the production approach for the product line. Software product line practice involves strategic, large-grained reuse.

Some practitioners use a different set of terms to convey essentially the same meaning. In this alternate terminology, a *product line* is a profit and loss center concerned with turning out a set of products; it refers to a business unit, not a set of products. The *product family* is that set of products, which we call the product line. The core asset base is called the *platform*. Previously, what we call core asset development was often referred to as *domain engineering* and what we call product development was referred to as *application engineering*.

The terminology is not as important as the concepts. That having been said, you might encounter different sets of terms in other places and should be able to translate between them.

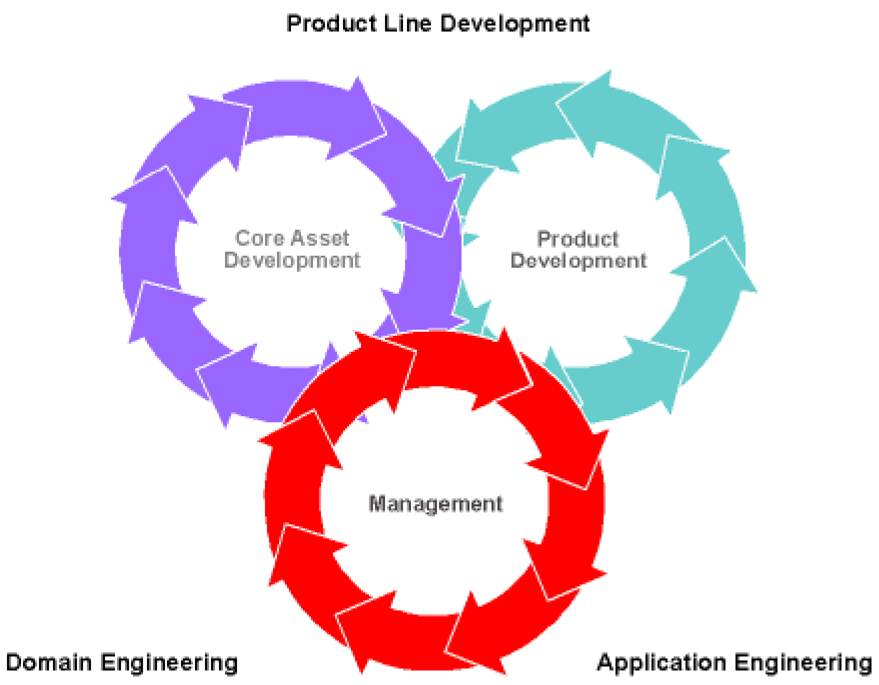
Starting Versus Running a Product Line

Many of the practice areas in this framework are written from the point of view of describing an in-place product line capability. We recognize that the framework will be used to help an organization put that capability in place, and ramping up to a product line is in many ways different than running one on a day-to-day basis.

We felt it was important to describe the end or "steady state" so that readers could understand the goals. However, to address the issues of starting (rather than running) a product line shop, the reader is referred to the "Launching and Institutionalizing" practice area.

Product Line Essential Activities

At its essence, fielding of a product line involves *core asset development* and *product development* using the core assets, both under the aegis of technical and organizational *management*. Core asset development and product development from the core assets can occur in either order: new products are built from core assets, or core assets are extracted from existing products. Often, products and core assets are built in concert with each other. The following figure illustrates this triad of essential activities.



The Three Essential Activities for Software Product Lines

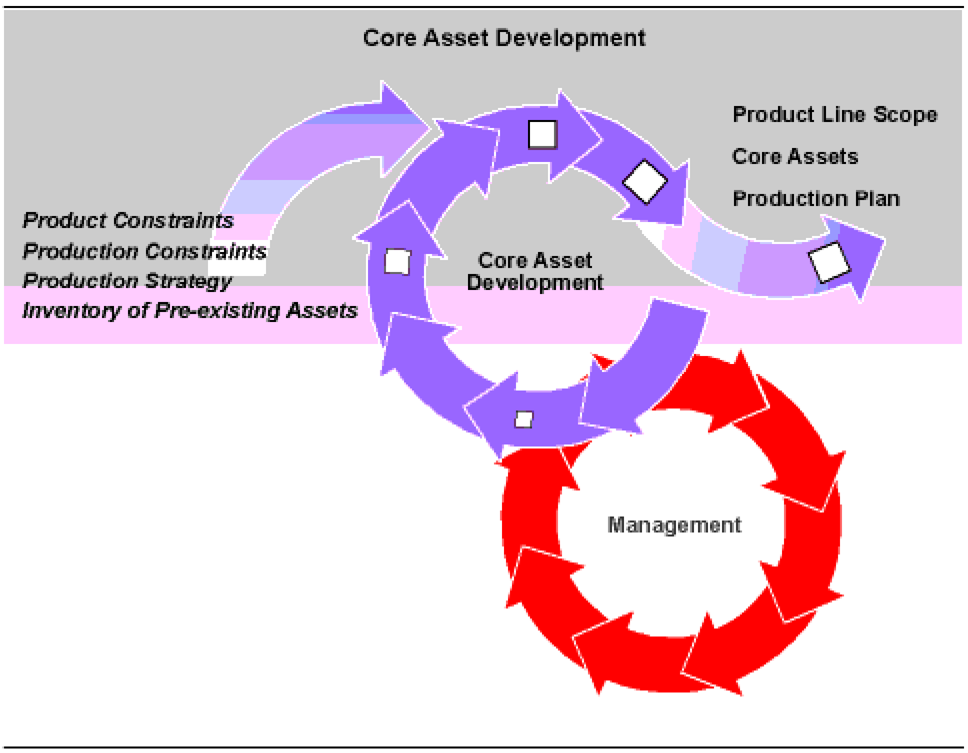
Each rotating circle represents one of the essential activities. All three are linked together and in perpetual motion, showing that all three are essential, are inextricably linked, can occur in any order, and are highly iterative.

The rotating arrows indicate not only that core assets are used to develop products, but also that revisions of existing core assets or even new core assets might, and most often do, evolve out of product development. The diagram in the above figure is neutral in regard to which part of the effort is launched first. In some contexts, already existing products are mined for generic assets—perhaps a requirements specification, an architecture, or software components—which are then migrated into the product line's core asset base. In other cases, the core assets may be developed or procured for later use in the production of products.

There is a strong feedback loop between the core assets and the products. Core assets are refreshed as new products are developed. Use of core assets is tracked, and the results are fed back to the core asset development activity. In addition, the value of the core assets is realized through the products that are developed from them. As a result, the core assets are made more generic by considering potential new products on the horizon. There is a constant need for strong, visionary management to invest resources in the development and sustainment of the core assets. Management must also precipitate the cultural change to view new products in the context of the available core assets. Either new products must align with the existing core assets, or the core assets must be updated to reflect the new products that are being marketed. Iteration is inherent in product line activities—that is, in turning out core assets, in turning out products, and in the coordination of the two. In the next three sections we examine the three essential activities in greater detail.

Core Asset Development

The goal of the core asset development activity is to establish a production capability for products. The following figure illustrates the core asset development activity along with its outputs and necessary inputs.



Core Asset Development

This activity, like its counterparts, is iterative. The rotating arrows suggest that there is no one-way causal relationship from inputs to outputs; the inputs and outputs of this activity affect each other. For example, slightly expanding the product line scope (one of the outputs) may admit whole new classes of systems to examine as possible sources of legacy assets (one of the inputs). Similarly, an input production constraint (such as mandating the use of a particular middleware product) may lead to restrictions on the architectural patterns (other inputs) that will be considered for the product line as a whole (such as the message-passing distributed object pattern). This restriction, in turn, will determine which preexisting assets are candidates for reuse or mining (still other inputs).

Three things are required for a production capability to develop products, and these three things are the outputs of the core asset development activity.

1. **Product line scope:**
   The product line scope is a description of the products that will constitute the product line or that the product line is capable of including. At its simplest, scope may consist of an enumerated list of product names. More typically, this description is cast in terms of the things that the products all have in common, and the ways in which they vary from one another. These might include features or operations they provide, performance or other quality attributes they exhibit, platforms on which they run, and so on.

Defining the product line scope is often referred to as *scoping*. For a product line to be successful, its scope must be defined carefully. If the scope is too large and product members vary too widely, then the core assets will be strained beyond their ability to accommodate the variability, economies of production will be lost, and the product line will collapse into the old-style one-at-a-time product development effort. If the scope is too small, then the core assets might not be built in a generic enough fashion to accommodate future growth, and the product line will stagnate: economies of scope will never be realized, and the full potential return on investment will never materialize.

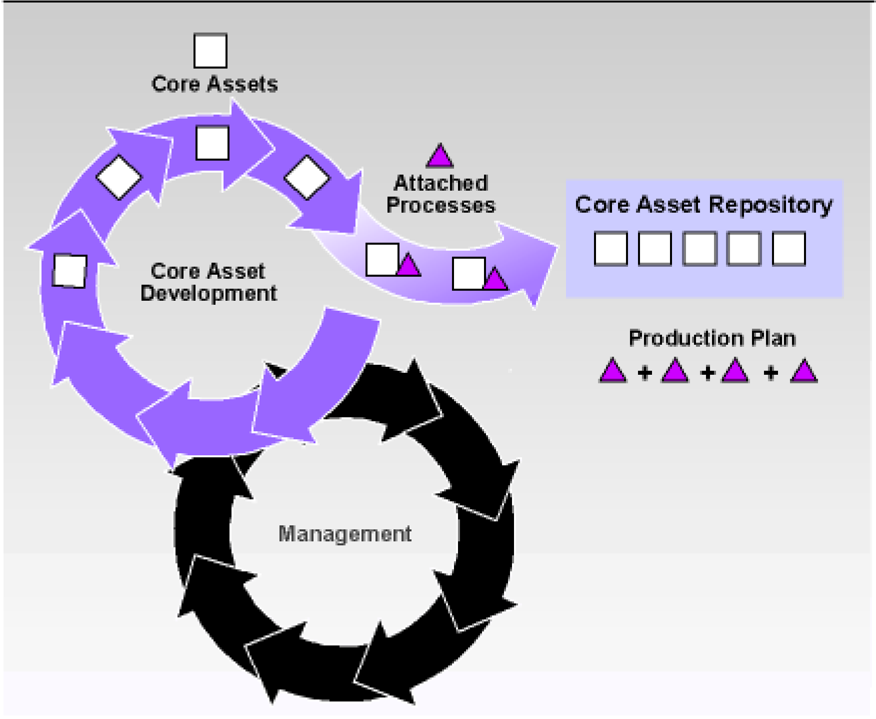
The scope of the product line must target the right products, as determined by knowledge of similar products or systems, the prevailing or predicted market factors, the nature of competing efforts, and the organization's business goals for embarking on a product line approach (such as merging a set of similar but currently independent product development projects).

The scope definition of a product line is itself a core asset, evolved and maintained over the product line's lifetime. Because it determines so much about the other core assets – in particular, what products they can support – we call it out separately.

2. **Core assets:**
Core assets are the basis for production of products in the product line. As we have already described, these core assets almost certainly include an architecture that the products in the product line will share, as well as software components that are developed for systematic reuse across the product line. Any real-time performance models or other architecture evaluation results associated with the product line architecture are core assets. Software components may also bring with them test plans, test cases, and all manner of design documentation. Requirements specifications and domain models are core assets, as is the statement of the product line's scope. Commercial off-the-shelf (COTS) software, if adopted, also constitute core assets. So do management artifacts such as schedules, budgets, and plans. Also, any production infrastructure such as domain-specific languages, tools, generators, and environments are core assets as well.

Among the core assets, the architecture warrants special treatment. A product line architecture is a software architecture that will satisfy the needs of the product line in general and the individual products in particular by explicitly admitting a set of variation points required to support the spectrum of products within the scope. The product line architecture plays a special role among the other core assets. It specifies the structure of the products in the product lines and provides interface specifications for the components that will be in the core asset base. Producing a product line architecture requires the product line scope (discussed above); a knowledge of relevant patterns and frameworks; and any available inventory of preexisting assets (all discussed below).

Each core asset should have associated with it an *attached process* that specifies how it will be used in the development of actual products. For example, the attached process for a set of product line requirements would give the process to follow when expressing the requirements for an individual product. This process might simply say: (1) use the product line requirements as the baseline requirements, (2) specify the variation requirement for any allowed variation point, (3) add any requirements outside the set of specified product line requirements, and (4) validate that the variations and extensions can be supported by the architecture. The process might also specify the automated tool support for accomplishing these steps. These attached processes are themselves core assets that get folded into what becomes the production plan for the product line. The following figure illustrates this concept of attached processes and how they are incorporated into the production plan.



Attached Processes

There are also core assets at a less technical level—namely, the training specific to the product line, the business case for use of a product line approach for this particular set of products, the technical management process definitions associated with the product line, and the set of identified risks for building products in the product line. Although not every core asset will necessarily be used in every product in the product line, all will be used in enough of the products to make their coordinated development, maintenance, and evolution pay off.

Finally, part of creating the core asset base is defining how that core asset base will be updated as the product line evolves, as more resources become available, as fielded products are maintained, and as technological changes or market shifts affect the product line scope.

3. **Production plan:**
   A production plan prescribes how the products are produced from the core assets. As noted above, core assets should each have an attached process that defines how it will be used in product development. The production plan is essentially a set of these attached processes with the necessary glue. It describes the overall scheme for how these individual processes can be fitted together to build a product. It is, in effect, the reuser's guide to product development within the product line. Each product in the product line will vary consistent with predefined variation points. How these variation points can be accommodated will vary from product line to product line. For example, variation could be achieved by selecting from an assortment of components to provide a given feature, by adding or deleting components, or by tailoring one or more components via inheritance or parameterization. It could also be the case that products are generated automatically. The exact vehicle to be used to provide the requisite variation among products is described in the production plan. Without the production plan, the product builder would not know the linkage among the core assets or how to utilize them effectively and within the constraints of the product line.

   To develop a production plan, you need to understand who will be building the products—the audience for the production plan. Knowing who the audience is will give you a better idea how to format the production plan. Production plans can range from a detailed process model to a much more informal guidebook. The degree of specificity required in the production plan depends on the background of the intended product builders, the structure of the organization, the culture of the organization, and the concept of operations for the product line. It will be useful to have at least a preliminary definition of the product line organization before developing the production plan.

   The production plan should describe how specific tools are to be applied in order to use, tailor, and evolve the core assets. The production plan should also incorporate any metrics defined to measure organizational improvement as a result of the product line (or other process improvement) practices and the plan for collecting the data to feed those metrics.

   For more information on production plans, see [Chastek 02b].

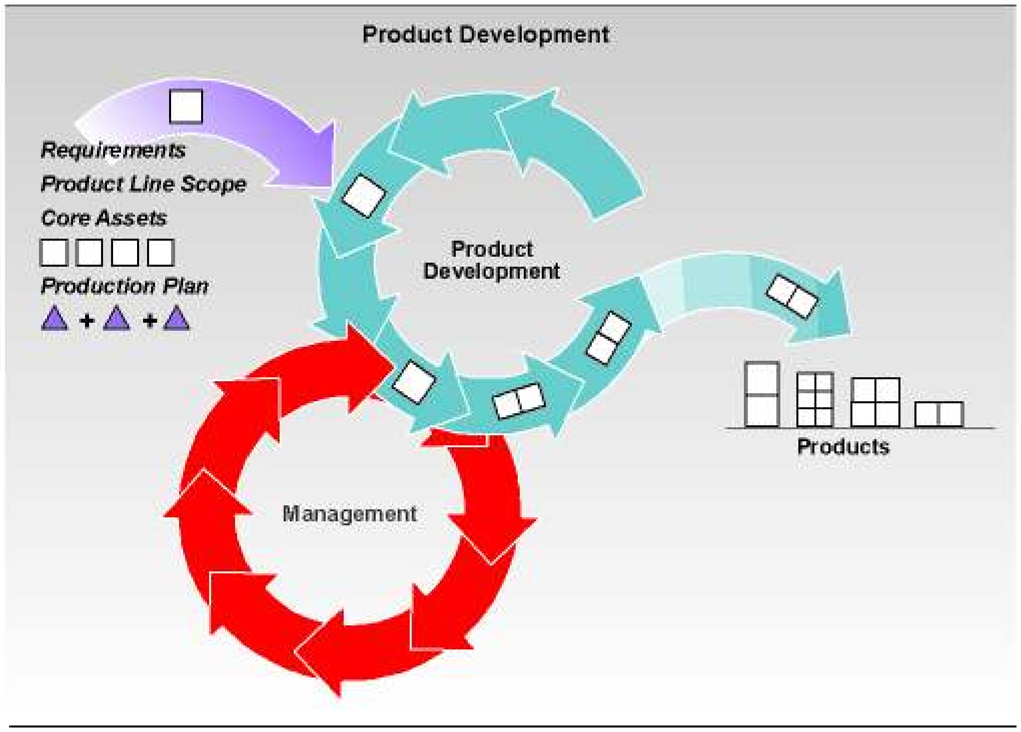
As will be seen in Product Development, these three outputs are necessary ingredients for feeding the product development activity, which turns out products that serve a particular customer or market niche.

The inputs to the core asset development activity are as follows.

1. **Product constraints:** What are the commonalities and variations among the products that will constitute the product line? What behavioral features do they provide? What features do the market and technology forecasts say will be beneficial in the future? What commercial, military, or company-specific standards apply to the products? What performance limits must they observe? With what external systems must they interface? What physical constraints must be observed? What quality requirements (such as availability and security) are imposed? The core assets must capitalize on the commonalities and accommodate envisioned variation with minimal tradeoff to product quality drivers such as security, reliability, usability, and so on. These constraints may be derived from a set of pre-existing products that will form the basis for the product line, or they may be generated anew, or some combination of two.
2. **Production constraints:** Must a new product be brought to market in a year, a month, or a day? What production capability must be given to engineers in the field? Answering these and similar questions will drive decisions about, for example, whether to invest in a generator environment or rely on manual coding. This in turn will drive decisions about what kind of variability mechanisms to provide in the core assets, and what form the overall production plan will take.
3. **Production strategy:** The production strategy is the overall approach for realizing the core assets and products. Will the product line be built proactively (starting with a set of core assets and spinning products off of them), reactively (starting with a set of products and generalizing their components to produce the product line core assets), or using some combination (see "All Three Together")? What will the transfer pricing strategy be—that is, how will the cost of producing the generic components be divided among the cost centers for the products? Will generic components be produced internally or purchased on the open market? Will products be automatically generated from the assets or will they be assembled? How will production of core assets be managed? The production strategy dictates the genesis of the architecture and associated components and the path for their growth.
4. **Inventory of preexisting assets:** Legacy systems embody an organization's domain expertise and/or define its market presence. The product line architecture, or at least pieces of it, may borrow heavily from proven structures of related legacy systems. Components may be mined from legacy systems. Such components may represent key intellectual property of the organization in relevant domains and therefore become prime candidates for components in the core asset base. What software and organizational assets are available at the outset of the product line effort? Are there libraries, frameworks, algorithms, tools, and components that can be utilized? Are there technical management processes, funding models, and training resources that can be easily adapted for the product line? The inventory includes all potential preexisting assets. Through careful analysis, an organization determines what is most appropriate to utilize. But preexisting assets are not limited to assets that were built by the product line organization. COTS and open-source products, as well as standards, patterns, and frameworks, are prime examples of preexisting assets that can be imported from outside the organization and used to good advantage.

Product Development

The product development activity depends on the three outputs described above—the product line scope, the core assets, and the production plan—plus the requirements for each individual product. The following figure illustrates these relationships.



Product Development

Once more, the rotating arrows indicate iteration and intricate relationships. For example, the existence and availability of a particular product may well affect the requirements for a subsequent product. As another example, building a product that has previously unrecognized commonality with another product already in the product line will create pressure to update the core assets and provide a basis for exploiting that commonality for future products.

The inputs for the product development activity are as follows:

- the requirements for a particular product, often expressed as a delta or variation from some generic product description contained in the product line scope (such a generic description is itself a core asset) or as a delta from the set of product line requirements (themselves a core asset).
- the product line scope, which indicates whether or not the product under consideration can be feasibly included in the product line
- the core assets from which the product is built
- the production plan, which details how the core assets are to be used to build the product

A software product line is, fundamentally, a set of related products, but how they come into existence can vary greatly depending on the core assets, the production plan, and the organizational context. From a very simple view, requirements for a product that is in the product line scope are received, and the production plan is followed so that the core assets can be properly used to develop the product. If the production plan is a more informal document, the product builders will need to build a product development plan that follows the guidance given. If the production plan is documented as a generic process description, the product builders will instantiate the production plan, recognizing the variation points being selected for the given product.

However, the process is rarely, if ever, so linear. The creation of products may have a strong feedback effect on the product line scope, the core assets, the production plan, and even the requirements for specific products. The ability to turn out a particular member of the product line quickly—perhaps a member that was not originally envisioned by the people responsible for defining the scope—will in turn affect the product line scope definition. Each new product may have similarities with other products that can be exploited by creating new core assets. As more products enter the field, efficiencies of production may dictate new system generation procedures, causing the production plan to be updated.

Management

Management plays a critical role in the successful fielding of a product line. Activities must be given resources, coordinated, and supervised. Management at both the technical (or project) and organizational levels must be strongly committed to the software product line effort. That commitment manifests itself in a number of ways that feed the product line effort and keep it healthy and vital. Technical management oversees the core asset development and to the product development activities by ensuring that the groups who build core assets and the groups who build products are engaged in the required activities, follow the processes defined for the product line, and collect data sufficient to track progress.

Organizational management must set in place the proper organizational structure that makes sense for the enterprise, and must make sure that the organizational units receive the right resources (for example, well-trained personnel) in sufficient amounts. We define organizational management as the authority that is responsible for the ultimate success or failure of the product line effort. Organizational management determines a funding model that will ensure the evolution of the core assets and then provides the funds accordingly. Organizational management also orchestrates the technical activities in and iterations between the essential activities of core asset development and product development. Management should ensure that these operations and the communication paths of the product line effort are documented in an operational concept. Management mitigates those risks at the organizational level that threaten the success of the product line. The organization's external interfaces also need careful management. Product lines tend to engender different relationships with an organization's customers and suppliers, and these new relationships must be introduced, nurtured, and strengthened. One of the most important things that management must do is create an adoption plan that describes the desired state of the organization (that is, routinely producing products in the product lines) and a strategy for achieving that state.

Both technical and organizational management also contribute to the core asset base by making available for reuse those management artifacts (especially schedules and budgets) used in developing products in the product line.

Finally, someone should be designated as the product line manager and that person must either act as or find and empower a product line champion. This person must be a strong, visionary leader who can keep the organization squarely pointed toward the product line goals, especially when the going gets rough in the early stages. Leadership is required for software product line success. Management and leadership are not always synonymous.

All Three Together

Each of the three activities—core asset development, product development, and management—is individually essential, and careful blending of all three is also essential—a blend of technology and business practices. Different organizations may take different paths through the three activities. The path they take is a manifestation of their production strategy, as described in "Core Asset Development."

Many organizations begin a software product line by developing the core assets first. These organizations take a *proactive* approach [Krueger 01]. They define their product line scope to define the set (more often, a *space*) of systems that will constitute their product line. This scope definition provides a kind of mission statement for designing the product line architecture, components, and other core assets with the right built-in variation points to cover the scope. Producing any product within that scope becomes a matter of exercising the variation points of the components and architecture—that is, configuring—and then assembling and testing the system. Other organizations begin with one or a small number of products they already have and from these generate the product line core assets and future products. They take a *reactive* approach.

Both of these approaches may be attacked iteratively. For example, a proactive approach may begin with the production of only the most important core assets, rather than all of them. Early products use those core assets. Subsequent products are built using more core assets as they are added to the collection. Eventually, the full core asset base is fielded; earlier products may or may not be reengineered to use the full collection. An iterative reactive approach works similarly; the core asset based is populated sparsely at first, using existing products as the source. More core assets are added as time and resources permit.

The proactive approach has obvious advantages—products come to market extremely quickly with a minimum of code-writing. But there are also disadvantages. It requires a significant up-front investment to produce the architecture and the components that are generic (and reliable) across the entire product space. And it also requires copious up-front predictive knowledge, something that is not always available. In organizations that have long been developing products in a particular application domain, this is not a tremendous disadvantage. For a *green field* effort, where there is no experience or existing products, this is an enormous risk.

The reactive approach has the advantage of a much lower cost of entry to software product lines because the core asset base is not built up front. However, for the product line to be successful, the architecture and other core assets must be robust, extensible, and appropriate to future product line needs. If the core assets are not built beyond the ability to satisfy the specific set of products already in the works, extending them for future products may prove too costly.

Product Line Practice Areas

Product Line Essential Activities of this framework introduced three essential activities that are involved in developing a software product line. These are (1) core asset development, (2) product development, and (3) management. This section defines in more detail what an organization must do to perform those broad essential activities. We do this by defining *practice areas*. A practice area is a body of work or a collection of activities that an organization must master to successfully carry out the essential work of a product line. Practice areas help to make the essential activities more achievable by defining activities that are smaller and more tractable than a broad imperative such as "Develop core assets." Practice areas provide starting points from which organizations can make (and measure) progress in adopting a product line approach for software.

So, to achieve a software product line you must carry out the three essential activities. To be able to carry out the essential activities you must master the practice areas relevant to each. By "mastering," we mean an ability to achieve repeatable, not just one-time, success with the work.

Almost all of the practice areas describe activities that are essential for any successful software development, not just software product lines. However, they all either take on particular significance or must be carried out in a unique way in a product line context. Those aspects that are specifically relevant to software product lines, as opposed to single-system development, will be emphasized.

Describing the Practice Areas

For each practice area we present the following information:

- An introductory overview of the practice area that summarizes what it's about. You will not find a definitive discourse on the practice area here, since in most cases there is overlap with what can be found in traditional software engineering and management reference books. We provide a few basic references if you need a refresher.

- Those aspects of the practice area that apply especially to a product line, as opposed to a single system. Here you will learn in what ways traditional software and management practice areas need to be refocused or tailored to support a product line approach.
- How the practice area is applied to core asset development and product development, respectively. We separate these two essential activities; although in most cases a given practice area applies to both of these broad areas, the lens that you look through to focus changes when you are building products versus developing core assets.
- A description of any *specific practices* that are known to apply to the practice area. A specific practice describes a particular way of accomplishing the work associated with a practice area. Specific practices are not meant to be end-to-end methodological solutions to carrying out a practice area but approaches to the problem that have been used in practice to build product lines. Whether or not a specific practice will work for your organization depends on context.
- Known risks associated with the practice area. These are ways in which a practice area can go wrong, to the detriment of the overall product line effort. Our understanding of these risks is borne out of the pitfalls of others in their product line efforts.
- A list of references for further reading, to support your investigation in areas where you desire more depth.

There are other kinds of information associated with each practice area, although these are not called out in the description. When planning to carry out the practice area, be sure to keep the following in mind:

- For each practice area, make a work plan for carrying it out. The work plan should specify the plan owner, specific tasks, who is responsible for doing them, what resources those people will be given, and when the results are due. More information about planning for product lines can be found in the "Technical Planning" and "Organizational Planning" practice areas.
- For each practice area, define metrics associated with tracking its completion and measuring its success. These metrics will help an organization identify where the practice areas are (or are not) being executed in a way that is meeting the organization's goals. More information about planning for measurement can be found in the "Data Collection, Metrics, and Tracking" practice area.
- Many practice areas produce tangible artifacts. For each practice area that does so, make a plan for keeping its produced artifacts up to date and identify the set of stakeholders who hold a vested interest in the artifacts produced. Collect organizational plans for artifact evolution and sustainment, and stakeholder definitions, in your product line's operational concept, which is discussed in the "Operations" practice area.
- Many practice areas lead to the creation of core assets of some sort. For those that do, define and document an attached process that tells how the core assets are used (modified, instantiated, and so on) to build products. These attached processes together form the production plan for the product line. The "Process

Definition" practice area describes the essential ingredients for defining these (and other) processes. The "Operations" and "Architecture Definition" practice areas describe documents for containing some of them.

Organizing the Practice Areas

Since there are so many practice areas, we need a way of organizing them for easier access and reference. We divide them loosely into three categories:

1. *Software engineering* practice areas are those necessary to apply the appropriate technology to create and evolve both core assets and products.
2. *Technical management* practice areas are those management practices necessary to engineer the creation and evolution of the core assets and the products.
3. *Organizational management* practice areas are those necessary for the orchestration of the entire software product line effort.

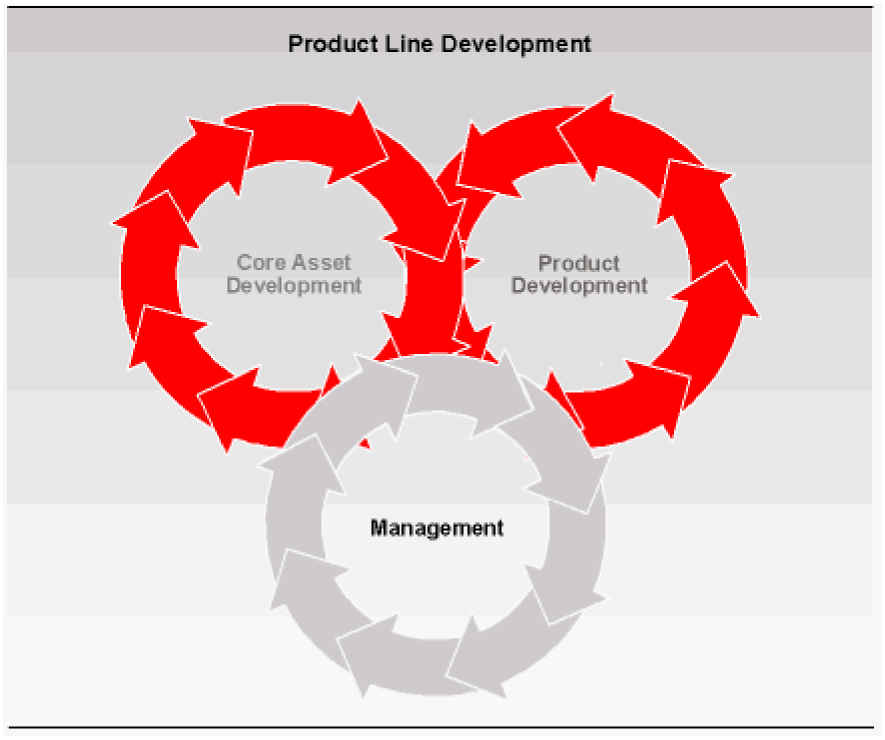
Each of these categories appeals to a different body of knowledge and requires a different skill set for the people needed to carry them out. The categories represent disciplines rather than job titles.

There is no way to divide cleanly into practice areas the knowledge necessary to achieve a software product line. Some overlap is inevitable. We have chosen what we hope to be a reasonable scheme and have identified practice area overlap where possible.

The description of practice areas that follows is an encyclopedia; neither the ordering nor the categorization constitutes a method or an order for application. In other works we provide product line practice patterns that show how to put the practice areas into play for a particular organization's context and goals [Clements 01a].

Software Engineering Practice Areas

Software engineering practice areas are those practice areas that are necessary for application of the appropriate technology to the creation and evolution of both core assets and products. They are carried out in the technical activities represented by the top two circles in the follow figure.

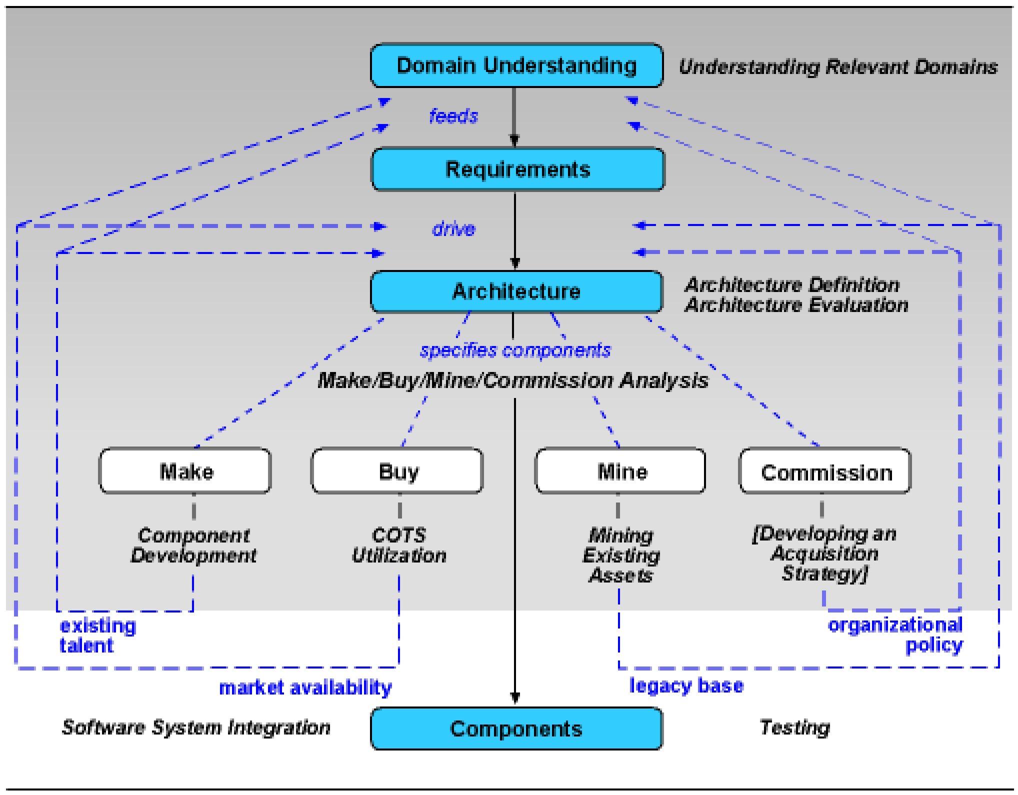


Software Engineering Practice Areas and the Essential Product Line Activities

In alphabetical order, they are:

- Architecture Definition
- Architecture Evaluation
- Component Development
- COTS Utilization
- Mining Existing Assets
- Requirements Engineering
- Software System Integration
- Testing
- Understanding Relevant Domains

All of these practice areas should sound familiar, because all are part of every well-engineered software system. But all take on special meaning when the software is a product line, as we will see. How do they relate to each other in a software product line context? The following figure sketches the story.



Relationships among Software Engineering Practice Areas[1]

Domain understanding feeds requirements, which drive an architecture, which specifies components. Components may be made in-house, bought on the open market, mined from legacy assets, or commissioned under contract. This choice depends on the availability of in-house talent and resources, open-market components, an exploitable legacy base, and able contractors. The existence (or nonexistence) of these things can affect the requirements and architecture for the product line. Once available, the components must be integrated, and they and the system must be tested. This is a quick trip through an iterative growth cycle, and it oversimplifies the story shamelessly but shows a good approximation of how the software engineering practice areas come into play.

We begin, fittingly, with architecture definition. Perhaps more than any other core asset, the architecture will determine how well an organization can field products that are built efficiently from a shared repository of core assets.

---

1. Items in brackets ([ ]) refer to practice areas other than those in the software engineering category.

Architecture Definition

This practice area describes the activities that must be performed to define a software architecture. By software architecture, we mean the following:

The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. "Externally visible" properties, we are referring to those assumptions other elements can make of an element, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on [Bass 03].

By making "externally visible properties" of elements[1] part of the definition, we intentionally and explicitly include elements' interfaces and behaviors as part of the architecture. We will return to this point later. By contrast, design decisions or implementation choices that do not have system-wide ramifications or visibility are not architectural.

Architecture is key to the success of any software project, not just a software product line. Architecture is the first design artifact that begins to place requirements into a solution space. Quality attributes of a system (such as performance, modifiability, and availability) are in large part permitted or precluded by its architecture—if the architecture is not suitable from the beginning for these qualities, don't expect to achieve them by some miracle later. The architecture determines the structure and management of the development project as well as the resulting system, since teams are formed and resources allocated around architectural elements. For anyone seeking to learn how the system works, the architecture is the place where understanding begins. The right architecture is absolutely essential for smooth sailing. The wrong one is a recipe for disaster.

Architectural requirements: For an architecture to be successful, its constraints must be known and articulated. And contrary to standard software engineering waterfall models, an architecture's constraints go far beyond implementing the required *behavior* of the system that is specified in a requirements document [Clements 01a, p. 57]. Other architectural drivers that a seasoned architect knows to take into account include:

- the quality attributes (as mentioned above) that are required for each product that will be built from the architecture
- whether or not the system will have to interact with other systems
- the business goals that the developing organization has for the system. These might include ambitions to use the architecture as the basis for other systems (or even other software product lines). Or perhaps the organization wishes to develop a particular competence in an area such as Web-based database access. Consequently, the architecture will be strongly influenced by that desire.
- best sources for components. A software architecture will, when it is completed, call for a set of components to be defined, implemented, and integrated. Those components may be implemented in-house (see the "Component Development" practice area), purchased from the commercial marketplace (see the "COTS

Utilization" practice area), contracted to third-party developers (see the "Developing an Acquisition Strategy" practice area), or excavated from the organization's own legacy vaults (see the "Mining Existing Assets" practice area). The architecture is indifferent as to the source. However, the availability of preexisting components (commercial, third-party, or legacy) may influence the architecture considerably and cause the architect to carve out a place in the architecture for a preexisting component to fit, if doing so will save time or money or play into the organization's long-term strategies.

Component interfaces: As we said in the opening of this section, architecture includes the interfaces of its components. It is therefore incumbent on the architect to specify those interfaces (or, if the component is developed externally, ensure that its interface is adequately specified by others). By "interface" we mean something far more complete than the simple functional signatures one finds in header files. Signatures simply name the programs and specify the numbers and types of their parameters, but they tell nothing about the semantics of the operations, the resources consumed, the exceptions raised, or the externally visible behavior. As Parnas wrote in 1971, an interface consists of the set of assumptions that users of the component may safely make about it—nothing more, but nothing less [Parnas 72]. Approaches for specifying component interfaces are discussed in "Specific Practices."

Connecting components: Applications are constructed by connecting together components to enable communication and coordination. In simple systems that run on a single processor, the venerable procedure call is the oldest and most widely used mechanism for component interaction. In modern distributed systems, however, something more sophisticated is desirable. There are several competing technologies, discussed below, for providing these connections as well as other infrastructure services. Among the services provided by the infrastructures are remote procedure calls (allowing components to be deployed on different processors transparently), communication protocols, object persistence and the creation of standard methods, and "naming services" that allow one component to find another via the component's registered name. These infrastructures are purchased as commercial packages; they are components themselves that facilitate connection among other components. These infrastructure packages are called middleware and, like patterns, represent another class of already solved problems (highly functional component interactions for distributed object-based systems) that the architect need not reinvent. At the time of this writing, the leading contenders in the race for middleware market dominance are the Object Management Group's CORBA [OMG 96]; Sun Microsystems' Java 2 Enterprise Edition (J2EE), including Enterprise Java Beans (EJB); and Microsoft's .NET.

Architecture documentation and views: Documenting the architecture is essential for it to achieve its effectiveness. Here, architectural views come into play. A view a representation of a set of system elements and the relationships among them [Clements 02a]. A view can be thought of as a projection of the architecture that includes certain kinds of information and suppresses other kinds. For example, a module decomposition view will show how the software for the system is hierarchically decomposed into smaller units of implementation. A communicating processes view will show the processes in the software and how they communicate or synchronize with each other, but it will not show how the software is divided into layers (if indeed it is). The layered view will show this, but will not show the processes. A deployment view shows how software is assigned to hardware elements in the system. There are many views of an architecture; choosing which ones to document is a matter of what information you wish to convey. Each view has a particular usefulness to one or more segments of the stakeholder community [IEEE 2000] and should be chosen and engineered with that in mind.

Aspects Peculiar to Product Lines

All architectures are abstractions that admit a plurality of instances; a great source of their conceptual value is, after all, the fact that they allow us to concentrate on design while admitting a number of implementations. But a product line architecture goes beyond this simple dichotomy between design and code; it is concerned identifying and providing mechanisms to achieve a set of explicitly allowed variations (because when exercised, these become products), whereas with a conventional architecture almost any instance will do as long as the (single) system's behavioral and quality goals are met. But products in a software product line exist simultaneously and may vary from each other in terms of their behavior, quality attributes, platform, network, physical configuration, middleware, scale factors, and a multitude of other ways.

In a conventional architecture, the mechanism for achieving different instances almost always comes down to modifying the code. But in a software product line, support for variation can take many forms (and be exercised at many times [Clements 01a, p. 64]). Mechanisms to achieve variation are discussed under "Specific Practices."

Integration may assume a greater role for software product lines than for one-off systems simply because of the number of times it's performed. A product line with a large number of products and upgrades requires a smooth and easy process for each product. Therefore, it pays to select a variation mechanism that allows for reliable and efficient integration when new products are turned out. This means some degree of automation. For example, if the variation mechanism chosen for the architecture is component selection and deselection, you will want an integration tool that carries out your wishes by selecting the right components and feeding them to the compiler or code generator. If the variation mechanism is parameterization or conditional compilation, you will want an integration tool that checks the parameter values for consistency and compatibility, then feeds those values to the compilation step. Hence, the variation mechanism chosen for the architecture will go hand-in-hand with the integration approach (see the "Software System Integration" practice area).

For many other system qualities, such as performance, availability, functionality, usability, and testability, there are no major peculiarities that distinguish architecture for product lines relative to one-of-a-kind systems.

There must be documentation for the product line architecture as it resides in the core asset base and for each product's architecture (to the extent that it varies from the product line architecture). For a software product line, the views will need to show the variations that are possible. A second documentation obligation is to describe the architecture's attached process-that is, the part of the production plan that deals with the architecture. It should describe the architecture's variation points, how to exercise them, and a rationale for the variation. In practice, the attached process for the architecture is often bundled with the attached processes for requirements engineering, component development, software integration, and testing into an operational document that serves as a product builder's guide, discussed in more detail under "Specific Practices."

Application to Core Asset Development

The product line architecture is an early and prominent member in the collection of core assets. The architecture is expected to persist over the life of the product line and to change relatively little and relatively slowly over time. The architecture defines the set of software components (and hence their supporting assets such as documentation and test artifacts) that populate the core asset base. The architecture also spawns its attached process, which is itself an important core asset for sustaining the product line.

Application to Product Development

Once it has been placed in the product line core asset base, the architecture is used to create instance architectures for each new product according to its attached process. If product builders discover a variation point or a needed mode of variation that is not permitted by the architecture, it should be brought to the architect's attention; if the variation is within scope (or deemed desirable to add to the scope), the architecture may be enhanced to accommodate it. The "Operations" practice area deals with setting up this feedback loop in the organization.

Specific Practices

Architecture definition and architecture-based development: As the field of software architecture has grown and matured, methods of creating, defining, and using architecture have proliferated. Many specific practices related to architecture definition are defined in widely available works [Kruchten 98, Jacobson 97, Hofmeister 00, Bachmann 00]. The Rational Unified Process (RUP) is the most widely used method for object-oriented systems. An explanation of RUP is beyond the scope of this framework, but a plethora of resources is available elsewhere; for example, at www.rational.com/products/rup/index.jsp.

Attribute-Driven Design (ADD): The SEI's Attribute-Driven Design (ADD) method [SEI ADD] is a method for designing the software architecture of a product line to ensure that the resulting products have the desired qualities. ADD is a recursive decomposition method that starts by gathering architectural drivers that are a combination of the quality, functional, and business requirements that "shape" the architecture.. The steps at each stage of the decomposition are:

1. Choose architectural drivers. The architectural drivers are the combination of quality, business, and functional goals that "shape" the architecture.
2. Choose patterns and children component types to satisfy drivers. There are known patterns to achieve various qualities. Choose the solutions that are most appropriate for the high priority qualities.
3. Instantiate children design elements and allocate functionality from use cases using multiple views. The functionality to be achieved by the product family is allocated to the component types.
4. Identify commonalities across component instances. These commonalities are what define the product line, as opposed to individual products.
5. Validate that quality and functional requirements and any constraints have not been precluded from being satisfied by the decomposition.
6. Refine use cases and quality scenarios as constraints to children design elements. Because ADD is a decomposition method, the inputs for the next stage of decomposition must be prepared.

Architectural patterns: Architectures are seldom built from scratch but rather evolve from solutions previously applied to similar problems. Architectural patterns represent a current approach to reusing architectural design solutions. An architectural pattern[2] is a description of component types and a pattern of their runtime control and/or data transfer [Shaw 96]. Architectural patterns are becoming a de facto design language for software architectures. People speak of pipe-and-filter, n-tier, client-server style, or an agent-based architectures, and these phrases immediately convey complex and sophisticated design information. Architectural pattern catalogues exist that explain the properties of a particular pattern, including how well-suited each one is for achieving specific quality attributes such as security or high performance. Using a previously catalogued pattern shortens the architecture definition process, because patterns come with pedigrees: what applications they work well for, what their performance properties are, where they can easily accommodate variation points, and so forth. For example, the layered architectural pattern is well known for imbuing a system with portability, the ability to move the software to a new operating environment with minimal change. Portability is a desirable characteristic for a product line architecture if different products are expected to run on different platforms, or if the entire product line may migrate to a new platform one day. Thus, a product line architect designing for portability may start out by considering a layered architecture. Product line architects should be familiar with well-known architectural patterns as well as patterns (well known or not) used in systems similar to the ones they are building.

Quality Attribute Workshops: Prerequisite to designing an architecture is understanding the behavioral and quality attribute requirements that it must satisfy. One way to elicit these requirements from the architecture's stakeholders is with a Quality Attribute Workshop (QAW) [SEI QAW]. QAWs provide a method for identifying a system's architecture critical quality attributes, such as availability, performance, security, interoperability, and modifiability. In the QAW, an external team facilitates meetings between stakeholders during which scenarios representing the quality attribute requirements are generated, prioritized, and refined (i.e., adding additional details such as the participants and assets involved, the sequence of activities, and questions about quality attributes requirements). The refined scenarios can be used in different ways, for example as seed scenarios for an evaluation exercise or as test cases in an acquisition effort.

Aspect-oriented programming (AOP): AOP is an approach to program development that makes it possible to modularize systemic properties of a program such as synchronization, error handling, security, persistence, resource sharing, distribution, memory management, replication, and the like. Rather than staying well localized within a class, these concerns tend to crosscut the system's class and module structure. An "aspect" is a special kind of module that implements one of these specific properties of a program. As that property varies, the effects "ripple" through the entire program automatically. Like object-oriented programming, AOP works by allowing the programmer to cleanly express certain structural properties of the program, and then take advantage of that structure in powerful ways. In object-oriented programming, the structure is rooted in notions of hierarchies, inheritance, and specialization. In AOP, the structure is rooted in notions of crosscutting. As an example, an AOP program might define "the public methods of a given package" as a crosscutting structure, and then say that all of those methods should do a certain kind of error handling. This would be coded in a few lines of well-modularized code. AOP is an architectural approach because it provides a means of separating concerns that would otherwise affect a multitude of components that were constructed to separate a different, orthogonal set of concerns. The AOP work at Xerox PARC is described on its Web page [Xerox 99].

Product builder's guide: A product line architecture is instantiated as a product architecture each time a product is turned out. The product architecture may be the same as the product line architecture, or it may be the result of preplanned tailoring or binding. For example, install *four* servers, 52 client workstations, and *two* databases; configure the network routers accordingly; use the *high-speed low-resolution version* of the graphics component, and turn encryption in the message generator *off*. The steps that product developers must take to create this product architecture constitute its attached process and should of course be documented as such. However, many organizations collect the attached processes for requirements engineering, architecture definition, component development, software integration, and testing into a single document that forms a specialized subset of the overall production plan. One organization we have worked with adopted the following organization for their product builder's guide:

- **Introduction:** goals and purpose of the document; intended audience; basic common assumptions; applicable development standards
- **Sources of other information:** references to documents containing the product line architecture definition (which is maintained separately from the product builder's guide because its stakeholders include more than product builders) and associated information such as terms and terminology, the architecture's goals,

architecture training materials, development standards, and configuration management procedures and policies

- **Basic concepts:** What is a variation point? What mechanisms for realizing variation points have been used in this architecture? What is the relation between the product line architecture and the architecture for a particular product? What is an architecture layer, and how is the concept used? What is a service (in this case, the basic unit of reuse provided by the architecture)? And so forth.
- **Service component catalogue:** This organization's product line architecture contains some preintegrated units of functionality called service components that product builders can use to construct products. This section catalogues those service components, defines their interfaces, and explains how service components related to each other.
- **Building an application:** This section gives code templates and examples for building applications. It progresses incrementally. First, how do you build the most trivial application possible, one that perhaps does nothing but start a process running? Then, how do you build the most trivial application that actually does something observable, the domain's equivalent of the ubiquitous "Hello, world!" program that was the first computer program many of us ever wrote? Then, how do you build an application that contains the functions common to many of the products in the product line? Then, how do you build an application that runs on a single processor? Distributed across multiple processors? And so forth. The examples show how to instantiate the architecture's variation points at each step along the way.
- **Performance engineering:** This section presented guidelines on how to build a product when performance was a concern.

Mechanisms for achieving variability in a product line architecture (1): Mikael Svahnberg and Jan Bosch have crisply staked out the landscape of architecture-based support for variability in product lines [Svahnberg 00]. Their list includes the following mechanisms:

- **Inheritance:** in object-oriented systems, used when a method needs to be implemented differently (or perhaps extended) for each product in the product line
- **Extensions and extension points:** used when parts of a component can be augmented with additional behavior or functionality
- **Parameterization:** used when a component's behavior can be characterized abstractly by a placeholder that is then defined at build time. Macros and templates are forms of parameterization.
- **Configuration and module interconnection languages:** used to define the build-time structure of a system, including selecting (or deselecting) whole components
- **Generation:** used when there is a higher-level language that can be used to define a component's desired properties

- **Compile-time selection of different implementations:** The variable *#ifdefs* can be used when variability in a component can be realized by choosing different implementations.

  Code-based mechanisms used to achieve variability within individual components will be discussed further in the "Component Development" practice area.

Mechanisms for achieving variability in a product line architecture (2): Philips Research Laboratories uses *service component frameworks* to achieve diversity in their product line of medical imaging systems [Wijnstra 00]. Goals for that family include extensibility over time and support for different functions at the same time. A framework is a skeleton of an application that can be customized to yield a product. White-box frameworks rely heavily on inheritance and dynamic binding; knowledge of the framework's internals is necessary in order to use it. Black-box frameworks define interfaces for components that can be plugged in via composition tools. A service component framework is a type of black-box framework, supporting a variable number of plug-in components. Each plug-in is a container for one or more services, which provide the necessary functionality. All services support the framework's defined interface but exhibit different behaviors. Clients use the functionality provided by the component framework and the services as a whole; the assemblage is itself a component in the products' architecture. Conversely, units in the product line architecture may consist of or contain one or more component frameworks.

Planning for architectural variation: Nokia has used a "requirements definition hierarchy" as a way to understand what variations are important to particular products [Kuusela 00]. The hierarchy consists of design objectives (goals or wishes) and design decisions (solutions adopted to meet the corresponding goals). For example, a design objective might be "System shall be highly reliable." One way to meet that objective is to decree that the "System shall be a duplicated system." This in turn might mean that the "System shall have duplicated hardware" and/or the "System duplicates communication links." Another way to meet the reliability objective is to decree that the "System shall have self-diagnostic capacity," which can be met in several ways. Each box in the hierarchy is tagged with a vector, each element of which corresponds to a product in the product line. The value of an element is the priority or importance given to that objective, or endorsement of that design decision, by the particular product. For example, if an overall goal for a product line is high reliability, being a duplicated system might be very important to Product 2 and Product 3, but not at all important to Product 1, which will be a single-chip system.

The requirements definition hierarchy is a tool that the architect can use as a bridge between the product line's scope (see the "Scoping" practice area), which will tell what variations the architecture will have to support, and the architecture, which may support the variation in a number of ways. It is also useful to see how widely used a new feature or variation will be: should it be incorporated into the architecture for many products to use, or is it a one-of-a-kind requirement best left to the devices of the product that spawned it? The hierarchy is a way for the architect to capture the rationale behind such decisions.

Architecture documentation: Recently more attention has been paid in the software engineering community about how to write down a software architecture so that others can understand it, use it to build systems, and sustain it. The Rational's Unified Modeling Language (UML) is the most-often used formal notation for software architectures, although it lacks many architecture-centric concepts. The Software Engineering Institute recently published the "views and beyond" approach to documentation [Clements 02a], which holds that documenting a software architecture is a matter of choosing the relevant views based on projected stakeholder needs, documenting those, and then documenting the information that applies across all of the views. Examples of cross-view information include how the views relate to each other, and stakeholder-centric roadmaps through the documentation that let people with different interests find information relevant to them quickly and efficiently. The approach includes a three-step method for choosing the best views to engineer and document for any architecture, and the overall approach produces a result compliant with the IEEE-recommended best practice on documenting architectures of software-intensive systems [IEEE 2000].

Specifying component interfaces: Interfaces are often specified using a contractual approach. Contracts state pre- and postconditions for each service and define invariants that express constraints about the interactions of services within the component. The contract approach is static and does not address the dynamic aspects of a component-based system or even the dynamic aspects of a single component's behavior. Additional techniques such as state machines [Harel 98] and interval temporal logic [Moszkowski 86] can be used to specify constraints on the component that deal with the ordering of events and the timing between events. For example, a service may create a thread and assign it work to do that will not be completed within the execution window of the service. A postcondition for that service would include the logical clause for "eventually this work is accomplished."

A complete contract should include information about what will be both provided and required. The typical component interface specification describes the services that a component provides. To fully document a component so that it can be integrated easily with other components, the specification should also document the resources that the component requires. In addition to making it easy to determine what must be available for the component to be integrated successfully, this documentation provides a basis for determining whether there are possible conflicts between the resources needed for the set of components comprising the application.

A component's interface provides only a specification of how individual services respond when invoked. As components are integrated, additional information is needed. The interactions between two components needed to achieve a specific objective can be described as a protocol. A protocol groups together a set of messages from both components and specifies the order in which they are to occur.

Each component exhibits a number of externally visible attributes that are important to its use but are often omitted (incorrectly) from its interface specification. Performance (throughput) and reliability are two such attributes. The standard technique for documenting the performance of a component is the computational complexity of the dominant algorithms. Although this technique is platform-independent, it is difficult to use in reasoning about satisfying requirements in real-time systems, because it fails to yield an actual time measure. Worse, it uses information that will change when algorithms (presumably encapsulated within the component) change. A better approach is to document performance bounds, setting an upper bound on time consumed. The documentation remains true when the software is ported to a platform at least as fast as the current one-a safe assumption in today's environment. Cases in which the stated bounds are not fast enough can be resolved on a case-by-case basis. If the product can in fact meet the more stringent requirement on that product's platform, that fact can be revealed. If it cannot, either remedial action must be taken or the requirement must be relaxed.

Practice Risks

The biggest risk associated with this practice area is failing to have a suitable product line architecture. This will result in:

- components that do not fit together or interact properly
- products that do not meet their behavioral, performance, or other quality goals
- products that should be in scope, but which are unable to be produced from the core assets at hand
- a tedious and ad hoc product-building process

These in turn will lead to extensive and time-consuming rework, poor system quality, and inability to realize the product line's full benefits. If product teams do not find the architecture to be suitable for their products and easy to understand and use, they may bypass it, resulting in the eventual degradation of the entire product line concept.

Unsuitable architectures could result from:

- **Lack of a skilled architect:** A product line architect must be skilled in current and promising technologies, the nuances of the application domains at hand, modern design techniques and tool support, and professional practices such as the use of architectural patterns. The architect must know all of the sources of requirements and constraints on the architecture, including those (such as organizational goals) not traditionally specified in a requirements specification [Clements 01a, p. 58].
- **Lack of sound input:** The product line scope and production strategy must be well defined and stable. The requirements for products must be articulated clearly and completely enough so that architectural decisions may be reliably based on them. Forthcoming technology, which the architecture must be poised to accept, must be forecast accurately. Relevant domains must be understood so that their

architectural lessons are learned. To the extent to which the architect is compelled to make guesses, the architecture poses a risk.

- **Poor communication:** The best architecture is useless if it is documented and communicated in ways that its consumers-the product builders-cannot understand. An architecture whose documentation is chronically out of date is effectively the same as an undocumented architecture. There must be clear and open two-way communication channels between the architect and the organizations using the architecture.
- **Lack of supportive management and culture:** There must be management support for the creation and use of the product line architecture, especially if the architecture group is separate from the product development group. Failing this, product groups may "go renegade" and make unilateral changes to the architecture, or decline to use it at all, when turning out their systems. There are additional risks if management does not support the strong integration of system and software engineering.
- **Architecture in a vacuum:** The exploration and definition of software architecture cannot take place in a vacuum separate from system architecture.
- **Poor tools:** There are precious few tools for this practice area, especially those that help with designing, specifying, or exercising an architecture's variability mechanisms–a fundamental part of a product line architecture. Tools to test the compliance of products to an architecture are virtually nonexistent.
- **Poor timing:** Declaring an architecture ready for production too early leads to stagnation, while declaring it too late may allow unwanted variation. Discretion is needed when deciding when and how firmly to freeze the architecture. The time required to fully develop the architecture also may be too long. If product development is curtailed while the product line architecture is being completed, developers may lose patience, management may lose resolve, and salespeople may lose market share.

Unsuitable architectures are characterized by:

- **Inappropriate parameterization:** Overparameterization can make a system unwieldy and difficult to understand. Underparameterization can eliminate some of the necessary customizations of the system. The early binding of parameters can also preclude easy customization, while the late binding of parameters can lead to inefficiencies.
- **Inadequate specifications:** Components may not integrate properly if their specifications are sketchy or limited to static descriptions of individual services.
- **Decomposition flaws:** A component may not provide the functionality needed to implement the system correctly if there is not an appropriate decomposition of the required system functionality.
- **Wrong level of specificity:** A component may not be reusable if the component is too specific or too general. If the component is made so general that it encompasses multiple domain concepts, the component may require complex configuration information to make it fit a specific situation and therefore be inherently difficult to reuse. The excessive generality may also tax performance

and other quality attributes to an unacceptable point. If the component is too specific, there will be few situations in which it is the correct choice.

- **Excessive intercomponent dependencies:** A component may become less reusable if it has excessive dependencies on other components.

Further Reading

General software architecture:

[Bass 03] emphasizes architecture's role in system development and provides several case studies of architectures used to solve real problems. One is an architecture for the CelsiusTech ship systems product line. It also includes an extensive discussion of architectural views.

[Shaw 96] provides an excellent treatment of architectural patterns (called styles there) and their ramifications for system building.

[Hofmeister 00] emphasizes views and structures, and provides a solid treatment of building a system from an architecture and its views.

[SEI ATA] provides a wide variety of software architecture resources and links.

Product line architecture:

[Bosch 00a] brings a dedicated product line focus to the mix, and is required reading for the product line practitioner.

Software architecture from a strictly object-oriented point of view:

[Booch 94] offers a good foundation.

[Jacobson 97] devotes an entire section to architectural patterns for object-oriented systems designed with strategic reuse in mind.

[Kruchten 98] is a good reference for the preeminent development process in the object-oriented realm.

[Buschmann 96] raises the design pattern phenomenon to the arena of software architecture and is a good staple of any architect's toolbox.

[Smith 01] contains three chapters of principles and guidance for architecting systems (object-oriented or not) in which performance is a concern.

Problem solving:

[Jackson 00] classifies, analyzes, and structures a set of recurring software development problems, organized according to how the software will interact with the outside world.

Architecture Documentation:

[Clements 02a] explains the views-and-beyond approach to architecture documentation.

UML:

(http://www.rational.com/uml) is the starting point.

---

1. An element is a unit of software that has identity at either implementation time or runtime. We use the term component to refer to a unit of software that serves as a core asset in the product line and that must be developed or acquired as a unit.
2. The term used in [Shaw 96] was architectural style, which is synonymous with architectural pattern.

Architecture Evaluation

"Marry your architecture in haste and you can repent in leisure." So admonished Barry Boehm in a recent lecture [Boehm 00]. The architecture of a system represents a coherent set of the earliest design decisions, which are the most difficult to change and the most critical to get right. It is the first design artifact that addresses the quality goals of the system such as security, reliability, usability, modifiability, and real-time performance. The architecture describes the system structure and serves as a common communication vehicle among the system stakeholders: developers, managers, maintainers, users, customers, testers, marketers, and anyone else who has a vested interest in the development or use of the system.

With the advent of repeatable, cost-effective architecture evaluation methods, it is now feasible to make architecture evaluation a standard part of the development cycle. And because so much rides on the architecture, and because it is available early in the life cycle, it makes utmost sense to evaluate the architecture early when there is still time for midcourse correction. In any nontrivial project, there are competing requirements and architectural decisions that must be made to resolve them. It is best to air and evaluate those decisions and to document the basis for making them before the decisions are cast into code.

Architecture evaluation is a form of artifact validation, just as software testing is a form of code validation. In the "Testing" practice area, we will discuss validation of artifacts in general—and in fact, prescribe a validation step for all of the product line's core assets—but the architecture for the product line is so foundational that we give its validation its own special practice area.

The evaluation can be done at a variety of stages during the design process. For example, the evaluation can occur when the architecture is still on the drawing board and candidate structures are being weighed. The evaluation can also be done later, after preliminary architectural decisions have been made, but before detailed design has begun. The evaluation can even be done after the entire system has been built (such as in the case of a reengineering or mining operation). The outputs will depend on the stage at which the evaluation is performed. Enough design decisions must have been made so that the achievement of the requirements and quality-attribute goals can be analyzed. The more architectural decisions that have been made, the more precise the evaluation can be. On the other hand, the more decisions that have been made, the more difficult it is to change them.

An organization's business goals for a system lead to particular behavioral requirements and quality-attribute goals. The architecture is evaluated with respect to those requirements and goals. Therefore, before an evaluation can proceed, the behavioral and quality-attribute goals against which an architecture is to be evaluated must be made explicit. These quality-attribute goals support the business goals. For example, if a business goal is that the system should be long-lived, modifiability becomes an important quality-attribute goal.

Quality-attribute goals, by themselves, are not definitive enough either for design or for evaluation. They must be made more concrete. Using modifiability as an example, if a product line can be adapted easily to have different user interfaces, but is dependent on a particular operating system, is it modifiable? The answer is yes with respect to the user interface, but no with respect to porting to a new operating system. Whether this architecture is suitably modifiable or not depends on what modifications to the product line are expected over its lifetime. That is, the abstract quality goal of modifiability must be made concrete: modifiable with respect to what kinds of changes, exactly? The same is true for other attributes. The evaluation method that you use must include a way to concretize the quality and behavioral goals for the architecture being evaluated.

Aspects Peculiar to Product Lines

In a product line, architecture assumes a dual role. There is the architecture for the product line as a whole, and there are architectures for each of the products. The latter are produced from the former by exercising the built-in variation mechanisms to achieve instances. Both should be evaluated. The product line architecture should be evaluated for its robustness and generality, to make sure it can serve as the basis for products in the product line's envisioned scope. Instance architectures should be evaluated to make sure they meet the specific behavioral and quality requirements of the product at hand. In practice, the extent to which these two evaluations are separate exercises depends on the extent to which the product architecture differs from the product line architecture. Evaluating both the product line and product architectures is a prudent, low-cost, risk-reduction method.

Some of the business goals will be related to the fact that the architecture is for a product line. For example, the architecture will almost certainly have built-in variation points that can be exercised to derive specific products having different attributes. The evaluation will have to focus on the variation points to make sure they are appropriate, offer sufficient flexibility to cover the product line's intended scope, can be exercised in a way that lets products be built quickly, and do not impose unacceptable runtime performance costs. Also, different products in the product line may have different quality-attribute requirements, and the architecture will have to be evaluated for its ability to provide all required combinations.

Often, some of the hardware and other performance-affecting factors for a product line architecture are unknown to begin with. Thus, the evaluation of the product line architecture must establish bounds on the performance that the architecture is able to achieve, assuming bounds on hardware and other variables. Use the evaluation to identify potential contention problems and to put in place the policies and strategies to resolve contention. The evaluation of a particular instance of the product line architecture can verify whether the hardware and performance decisions that have been made are compatible with the goals of that instance.

Application to Core Asset Development

Clearly, an evaluation should be applied to the core asset that is the product line architecture. As the requirements, business goals, and architecture all evolve over time, there should be periodic (although not frequent) mini-evaluations that discover whether the architecture and business goals are still well matched. Some evaluation methods produce a report that summarizes what the articulated, prioritized quality-attribute goals are for the architecture, and how the architecture satisfies them. Such a report makes an excellent rationale record, which can then accompany the architecture throughout its evolution as a core asset in its own right.

An architecture evaluation can also be performed on components that are candidates to be acquired as core assets, as well as on components developed in-house. In either case, the evaluation proceeds with technical personnel from the organization that developed the potential acquisition. An architecture evaluation is not possible for "black-box" architecture acquisitions where the architecture is not visible. The quality-attribute goals to be used for the evaluation will include how well the potential acquisition will (1) support the quality goals for the product line and (2) evolve over time to support the intended evolution of the products in the product line.

Application to Product Development

An architecture evaluation should be performed on an instance or variation of the architecture that will be used to build one or more of the products in the product line. The extent to which this is a separate, dedicated evaluation depends on the extent to which the product architecture differs in quality-attribute-affecting ways from the product line architecture. If it doesn't, then these product architecture evaluations can be abbreviated, since many of the issues that would normally be raised in a single product evaluation will have been dealt with in the evaluation of the product line architecture. In fact, just as the product architecture is a variation of the product line architecture, the product architecture evaluation is a variation of the product line architecture evaluation.

Therefore, depending on the architecture evaluation method used, the evaluation artifacts (scenarios, checklists, and so on) will certainly have reuse potential, and you should create them with that in mind. Document a short attached process for the architecture evaluation of the product line or product architectures. This process description would include the method used, what artifacts can be reused, and what issues to focus on. The results of architecture evaluation for product architectures often provide useful feedback to the architect(s) of the product line architecture and fuel improvements in the product line architecture.

Finally, when a new product is proposed that falls outside the scope of the original product line (for which the architecture was presumably evaluated), the product line architecture can be reevaluated to see if it will suffice for this new product. If it will, the product line's scope is expanded to include the new product. If it will not, the evaluation can be used to determine how the architecture would have to be modified to accommodate the new product.

Specific Practices

Several different architecture evaluation techniques exist and can be modified to serve in a product line context. Techniques can be categorized broadly as either questioning techniques (those using questionnaires, checklists, scenarios, and the like as the basis for architectural investigation) or measuring techniques (such as simulation or experimentation with a running system) [Abowd 96]. A well-versed architect should have a spectrum of techniques in his or her evaluation kit. For full-fledged architectures, software performance engineering or a method such as the ATAM^{SM} or the SAAM is indispensable. For less fully worked out designs, a technique such as Active Reviews for Intermediate Designs (ARID) is handy. For validating architectural (and other design) specifications, active design reviews (ADRs) are helpful. A bibliography of software architecture analysis, available from the journal *Software Engineering Notes* [Zhao 99], provides more alternatives.

ATAM^{SM}: The Architecture Tradeoff Analysis Method^{SM} (ATAM) is a scenario-based architecture evaluation method that focuses on a system's quality goals. The input to the ATAM consists of an architecture, the business goals of a system or product line, and the perspectives of stakeholders involved with that system or product line. The ATAM achieves its evaluation of an architecture by utilizing an understanding of the architectural approach that is used to achieve particular quality goals and the implications of that approach. The ATAM utilizes stakeholder perspectives to derive a collection of scenarios giving specific instances for usage, performance requirements, various types of failures, possible threats, and a set of likely modifications. The scenarios are used for the evaluators to understand the inherent architectural risks, sensitivity points to particular quality attributes, and tradeoffs among quality attributes. Of particular interest to ATAM-based evaluations of product line architectures are the sensitivity points to extensibility (or variation) and the tradeoffs of extensibility with other quality-attribute goals (usually real-time performance, security, and reliability).

The output of an ATAM evaluation includes:

- the collection of scenarios that represent the stakeholders' highest-priority expression of usage and quality-attribute goals for the system and its architecture
- a utility tree that assigns specific scenarios to a location in the "space" of quality attributes that apply to the system(s) whose architecture is being evaluated
- specific analysis results, including the explicit identification of sensitivity points, tradeoffs, and other architectural decisions that impact desired quality attributes either positively or problematically. The latter constitute areas of risk.

The ATAM can be used to evaluate both product line and product architectures at various stages of development (conceptual, before code, during development, or after deployment). An ATAM evaluation usually requires three full days plus some preparation and preliminary investigation time. The ATAM is described fully by Clements et al. [Clements 01b] and on the World Wide Web [SEI ATA].

SPE: Software performance engineering (SPE) is a method for making sure that a design will allow a system to meet its performance goals before it has been built. SPE involves articulating the specific performance goals, building coarse-grained models to get early ideas about whether the design is problematic, and refining those models along well-defined lines as more information becomes available. Conceptually, SPE resembles the ATAM, in which the singular quality of interest is performance. Connie Smith has written both the definitive resource for SPE and its concise method description [Smith 90, Smith 99].

ARID: Active Reviews for Intermediate Designs (ARID) [Clements 00] is a hybrid design review method that combines the active design review philosophy of ADRs with the scenario-based analysis of the ATAM and SAAM. ARID was created to evaluate partial (subsystem, for example) designs in their early or conceptual phases, before they are fully documented. While such designs are architectural in nature, they are not complete architectures. ARID works by assembling stakeholders for the design, having them adopt a set of scenarios that express a set of meaningful ways they would want to use the design, and then having them write code or pseudocode that uses the design to carry out each scenario. This will wring out any conceptual flaws early, plus give stakeholders an early familiarity with the design until it is completely documented. An ARID exercise takes from one to two days.

Active design reviews: An Active Design Review (ADR) [Parnas 85] is a technique that can be used to evaluate an architecture still under construction. ADRs are particularly well-suited for evaluating the designs of single components or small groups of components before the entire architecture has been solidified. The principle behind ADRs is that stakeholders are engaged to review the documentation that describes the interface facilities provided by a component, but the stakeholders are asked to complete exercises that compel them to actually use the documentation. For example, each reviewer may be asked to write a short code segment that performs some useful task using the component's interface facilities, or each reviewer may be asked to verify that essential information about each interface operation is present and well-specified. ADRs are contrasted with unstructured reviews in which people are asked to read a document, attend a long meeting, and comment on whatever they wish. In an ADR, there is no meeting; reviewers are debriefed (or walked through their assignments) individually or in small informal groups. The key is to avoid asking questions to which a reviewer can blithely and without much thought answer "yes" or "no." An ADR for a medium-sized component usually takes a full day from each of about a half dozen reviewers who can work in parallel. The debriefing takes about an hour for each session.

Practice Risks

The major risk associated with this practice is failing to perform an effective architecture evaluation that will prevent unsuitable architectures from being allowed to pollute a software product line effort. Architecture evaluation is the safety valve for product line architectures, and an ineffective evaluation will lead to the same consequences as an unsuitable architecture, which were listed in the "Architecture Definition" practice area.

An ineffective evaluation can result from the following:

- **Wrong people involved in the evaluation:** If the architect is not involved in the evaluation, it is unlikely that enough information will be uncovered to make the evaluation worthwhile. Similarly, if the architecture's stakeholders are not involved, the comprehensive goals and requirements for the architecture (against which it must be evaluated) will not emerge.
- **Wrong time in the life cycle:** If the review is too early, not enough decisions have been made, so there isn't anything to evaluate. If the review is too late, little can be changed as a result of the evaluation.
- **No time for evaluation:** If time is not planned for the evaluation, the people who need to be involved will not be able to give it their attention, the evaluation will not be conducted effectively, and the results will be superficial at best.
- **Wrong interpretation of evaluation:** The results of any architecture evaluation should not be seen as a complete enumeration of all of the risks in the development. Process deficiencies, resource inadequacies, personnel issues, and downstream implementation problems are all risks unlikely to be exposed by an architecture evaluation.
- **Failure to reevaluate:** As the architecture inevitably evolves, or the criteria for its suitability inevitably evolve, it should be reevaluated (perhaps using a lightweight version of the original evaluation) periodically to give the organization confidence that they are on the right track.

Further Reading

[Clements 01b] is a primer on software architecture evaluation, containing a detailed process model and practical guidance for applying the ATAM and comparing it with other evaluation methods. Other methods, including ARID, are also covered.

[Parnas 85] is the original description by Parnas and Weiss of ADRs and remains the most comprehensive source of information on this approach.

[SEI ATA], the home page for the Software Engineering Institute's Software Architecture Technology (formerly known as the Architecture Tradeoff Analysis) Initiative, contains publications about the ATAM and the SAAM, as well as other software architecture topics.

[Smith 90] remains the definitive treatment of performance engineering.

[Smith 01] is a good accompaniment, but not a substitute, for [Smith 90].

[Zhao 99] has compiled a bibliography on software architecture analysis. His Web site, where the list is kept up to date, is cited on the SEI's ATA page.

Component Development

One of the tasks of the software architect is to produce the list of components that will populate the architecture. This list gives the development, mining, and acquisition teams their marching orders for supplying the parts that the software system will comprise. The term "component" is about as generic as the term "object"; definitions for each term abound. Simply stated, components are the units of software that go together to form whole systems (products), as dictated by the software architecture for the products. Szyperski offers a more precise definition that applies well [Szyperski 98]:

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

By component development, we mean the production of components that implement specific functionality within the context of a software architecture. The functionality is encapsulated and packaged, then integrated with other components using an interconnection method.

Software components trace their heritage back to the subroutine, which was the first unit of software reuse. Programmers discovered they could invoke a previously written segment of code and have access to its functionality while being blissfully unconcerned with its implementation, development history, storage management, and so forth. Soon, very few people ever again had to worry about how to code, say, a numerically stable double-precision cosine algorithm. Besides saving time, this practice elevated our thinking: we could think "cosine" and not about storage registers and overflowing multiplications. It also elevated our languages: sophisticated subroutines were indistinguishable from primitive, atomic statements in the programming language.

What we now call component-based software development flows in an unbroken line from these early beginnings. Modern components are much larger, are much more sophisticated, carry us much higher into domain-specific application realms, and have more complex interaction mechanisms than subroutine invocation, but the concepts and the reasons we embrace the concepts remain the same. In the same way that early subroutines liberated the programmer from thinking about details, component-based software development shifts the emphasis from *programming software to composing software systems*. Implementation has given way to integration as the focus. At its foundation is the assumption that there is sufficient commonality in many large software systems to justify developing reusable components to exploit and satisfy that commonality. Today, we look for components that provide large collections of related functionality all at once (instead of a cosine routine, think *Mathematica*) and whose interconnections with each other are loose and flexible. If we have control over the decomposition into components and the interfaces of each, then the granularity and interconnection is determined by our system's software architecture. If the components are built externally, then their granularity and interfaces are imposed on us, and they affect our software architecture.

The practice area of component development is concerned with the former case, and how to build the components so that the instructions given to us in the architecture are carried out. (One type of instruction carried out by the architecture is what other parts of a system a component is allowed to use. A layered view of an architecture, for example, is highly concerned with this. Allowed-to-use information is not part of the component's interface or of its functionality, but it is nevertheless architectural and must be honored by the implementation.) Very complex components may have substructure of their own and be implemented partially by employing smaller components, either built or acquired.

Aspects Peculiar to Product Lines

For the purposes of product lines, components are the units of software that go together to form whole systems (products), as dictated by the product line architecture for the products and the product line as a whole. If we appeal to the Szyperski definition of components given above, "deployed independently" may simply mean installed into a product line's core asset base where they are made available for use in one or more products. The "third parties" are the product developers, who compose the component with others to create systems. The contractually specified interfaces are paramount, as they are in any software development paradigm with software architecture at its foundation.

The component development portion of a product line development effort focuses on providing the operational software that is needed by the products and that is to be developed in-house. The resultant components either are included in the core asset base and hence used in multiple products in the product line or are product-specific components. Components that are included in the core asset base must support the flexibility needed to satisfy the variation points specified in the product line architecture and/or the product line requirements. Needed functionality is defined in the context of the product line architecture. The architecture also defines those places at which variation is needed.

The singular aspect of component development that is peculiar to product lines is providing required variability in the developed components via the mechanisms that are described in the specific practices for this practice area.

Application to Core Asset Development

If a developed component is to be a core component, it must have an attached process associated with it that explains how any built-in component-level variability can be exercised in order to produce an instantiated version for a particular product. Developed components and their related artifacts (interface specifications, attached processes for instantiating built-in variability, test support, and so on) constitute a major portion of the product line's core asset base. Hand in hand with the software architecture that mandated them into existence, the core components form the conceptual basis for building products. Consequently, component development, as described above, is a large portion of the activity on the core asset development side of product line operations.

Application to Product Development

If a developed component is not to be part of the core asset base, this suggests that it is specific to a particular product and therefore probably does not have much variability built into it. While the development task must obey the architecture as strictly as it must for core components, noncore development is likely to be simpler. Nevertheless, developers of noncore components would be wise to look for places where variability could be installed in the future, should the component in question ever turn out to be useful in a group of products.

Components for a product are (1) used directly from the core asset base, (2) used directly after binding the built-in variabilities, (3) used after modification or adaptation, or (4) developed anew. Since the first two cases are pro forma, we will discuss the last two.

Adapting components: Components that are being used in a context other than the one for which they were originally developed often do not exactly fit their assigned roles. There are a couple of techniques for accommodating these differences. The adapter design pattern [Gamma 95] imposes an intermediary between two components. The adapter can compensate for mismatches in number or types of parameters within a service signature, provide synchronization in a multithreaded interaction, and adjust for many other types of incompatibilities. Scripting languages can often be used to implement the adapter.

The second technique is to modify the component to fit its new environment. This may be impossible if the source code is not available. Even if it is possible, it is usually a bad idea. Cloning an existing component creates a new asset that must be managed and creates a dependency that cannot be expressed explicitly. It can vary independently of its parent component, making maintenance of both pieces a difficult task. Object-oriented notations provide a semantic device to express this type of relationship by defining the dependent class in terms of an extension of the original class. Although similar devices do not exist at the component level, a new component may be implemented by deriving objects from those that implement the original component.

Developing new components: New development should occur only after a thorough search has been made of existing core assets. In some organizations, the product team may have to "contract" with a component development organization to build the needed component. If it is built in the product organization, there should be product line standards to follow for the creation of the core assets supporting the component.

Whether a product component is adapted or built from scratch, it should be reviewed ultimately for "promotion" to the core asset base (and, in fact, should be developed with that in mind). To help with that review, robustness analysis [Jacobson 97] can be applied to determine how flexible the product is with respect to future changes in requirements. By examining change cases (use cases that are not yet requirements), the team identifies points in the system that would need changes in order to support the new requirements. This provides a feedback loop to the component developers. Specifications for new components and modifications to existing ones are the outputs of this analysis.

Specific Practices

The specific practices in this practice area all deal with component-level variability mechanisms.

Variability mechanisms (1): Jacobson et al., discuss the mechanisms for supporting variability in components, which are shown in the table below [Jacobson 97]. Each mechanism provides a different type of variability. The variation of functionality happens at different times depending on the type. Some of these variation types are included in the specification implicitly. For example, when a parameter is used, the specification is taken to include the specific type of component mentioned in the contract or any component that is a specialization of that component. In the template instantiation example in the table, the parameter to the template is Container, which permits variation implicitly via the inheritance pattern. The Container parameter can be replaced by any of its subclasses, such as Set or Bag.

Types of Variation [Jacobson 97]

Mechanism	**Time of Specialization**	**Type of Variability**
Inheritance	At class definition time	Specialization is done by modifying or adding to existing definitions. Example: LongDistanceCall inherits from PhoneCall.
Extension	At requirements time	One use of a system can be defined by adding to the definition of another use. Example: WithdrawalTransaction extends BasicTransaction.
Uses	At requirements time	One use of a system can be defined by including the functionality of another use.

		Example: WithdrawalTransaction uses the Authentication use.
Configuration	Previous to runtime	A separate resource, such as file, is used to specialize the component. Example: JavaBeans properties file
Parameters	At component implementation time	A functional definition is written in terms of unbound elements that are supplied when actual use is made of the definition. Example: calculatePriority(Rule)
Template instantiation	At component implementation time	A type specification is written in terms of unbound elements that are supplied when actual use is made of the specification. Example: ExceptionHandler<Container>
Generation	Before or during runtime	A tool that produces definitions from user input. Example: Configuration wizard

Variability can also be shown explicitly, but that is more cumbersome than the implicit approach. The javadoc tool in Java lists all of the known subclasses of the class whose documentation is being created. This requires that the documentation for the parent class be regenerated every time a new subclass is declared. Any explicit listing of variants will require this type of maintenance. The variations may also be captured in an activity diagram that maps alternative paths.

One aspect of variability that is important in a product line effort is whether the variants must be identified at the time of product line architecture definition or can be discovered during the individual product's architectural phase. Inheritance allows for a variant to be created without the existing component having knowledge of the new variant. Likewise, template instantiation allows for the discovery of new parameter values after the template is designed; however, the new parameter must satisfy the assumptions of the template, which may not be stated explicitly in the interface of the formal parameter. In most cases, configuration further constrains the variation to a fixed set of attributes and a fixed set of values for each attribute.

Variability mechanisms (2): Anastasopoulos and Gacek expound a somewhat different set of variability options [Anastasopoulos 00]. Their list includes:

- Aggregation/delegation, an object-oriented technique in which functionality of an object is extended by delegating work it cannot normally perform to an object that can. The delegating object must have a repertoire of candidates (and their methods) known to it and assumes a role resembling that of a service broker.
- Inheritance, which assigns base functionality to a superclass and extended or specialized functionality to a subclass. Complex forms include dynamic and multiple inheritance, in addition to the more standard varieties.

- Parameterization, as described above.
- Overloading, which means reusing a named functionality to operate on different types. Overloading promotes code reuse, but at the cost of understandability and code complexity.
- Properties in the Delphi language, which are attributes of an object. Variability is achieved by modifying the attribute values or the actual set of attributes.
- Dynamic class loading in Java, where classes are loaded into memory when needed. A product can query its context and that of its user to decide at runtime which classes to load.
- Static libraries, which contain external functions that are linked to after compilation time. By changing the libraries, one can change the implementations of functions whose names and signatures are known.
- Dynamic link libraries, which give the flexibility of static libraries but defer the decision until runtime based on context and execution conditions.
- Conditional compilation puts multiple implementations of a module in the same file, with one chosen at compile-time by providing appropriate preprocessor directives.
- Frame technology. Frames are source files equipped with preprocessor-like directives that allow parent frames to copy and adapt child frames and form hierarchies. On top of each hierarchical assembly of frames lies a corresponding specification frame that collects code from the lower frames and provides the ready-to-compile module that results.
- Reflection, the ability of a program to manipulate data that represents information about itself or its execution environment or state. Reflective programs can adjust their behavior based on their context.
- Aspect-oriented programming, which was described in the "Architecture Definition" practice area.
- Design patterns, which are extensible, object-oriented solution templates catalogued in various handbooks (for example [Gamma 95]). The adapter pattern was mentioned specifically as a variability mechanism earlier in this practice area.

Practice Risks

The overriding risk in component development is building unsuitable components for the software product line applications. This will result in poor product quality, the inability to field products quickly, low customer satisfaction, and low organizational morale. Unsuitable components can come about by:

- **Not enough variability:** Components not only must meet their behavioral and quality requirements (as imposed on them by the product line's software architecture) but also must be tailorable in preplanned ways to enable product developers to instantiate them quickly and reliably in the correct forms for specific products.
- **Too much variability:** Building in too much variability can prevent the components from being understood well enough to be used effectively, or can cause unforeseen errors when the variabilities conflict with each other.

- **Choosing the wrong variation mechanism(s) for the job:** The wrong choice can result in components that cannot be tailored at the time they need to be.
- **Poor quality of components:** Components of poor quality will set back any effort, but poor core asset components will undermine the entire product line. Product builders will lose confidence with the core asset builders, and pressure to bypass them will mount. The "Testing" practice area should be applied to ameliorate this risk.

Further Reading

[Szyperski 98]: Szyperski provides a comprehensive presentation on components. It provides a survey of component models and covers supporting topics such as domain analysis and component frameworks.

[Jacobson 97] and [Anastasopoulos 00]: These two works together provide a superb compendium of component-level variability mechanisms that are available to a product line component developer.

Mining Existing Assets

Mining existing assets refers to resurrecting and rehabilitating a piece of an old system to serve in a new system for which it was not originally intended. Often it simply refers to finding useful legacy code from an organization's existing systems portfolio and reusing it within a new application. However, the code-only view completely misses the big picture. We have known for years that in the grand scheme of things, code plays a small role in the cost of a system. Coding is simply not what's difficult about system/software development. Rich candidates for mining include a wide range of assets besides code—assets that will pay lucrative dividends. Business models, rule bases, requirements specifications, schedules, budgets, test plans, test cases, coding standards, algorithms, process definitions, performance models, and the like are all wonderful assets for reuse. The only reason so-called "code reuse" pays at all is because of the designs and algorithms and interfaces that come along with the code [Clements 01a, p. 99].

For example, whole or partial architectures, and the design decisions they embody (captured by documented rationale) are especially valuable. And if a mined architecture is suitable, then probably the components that originally populated it can be migrated along with it. But to determine fitness for reuse of either the architecture or its components, it is necessary to obtain a thorough architectural understanding of the legacy system. And, of course, the architect may be long gone. If good documentation does not exist, the process of architecture reconstruction may need to be employed. Reconstruction will reveal the interactions and relations among the architecture's components. It will illuminate constraints for how, if mined, the components can interact within the architecture of the new or updated software. It can also help to understand the tradeoff options available for reusing components in a new or improved way [Kazman 02, O'Brien 02]. Once the architecture has been extracted, it can be evaluated for suitability using the techniques described in the "Architecture Evaluation" practice area.

Documentation is an asset that is often overlooked and may have significant reuse potential. Much of the corporate knowledge about the software assets may be captured in the existing legacy documentation assets. This makes these documentation assets highly desirable candidates for mining and rehabilitation, especially where the associated software assets are being mined and rehabilitated and they closely correlate with one another.

Mining involves understanding of what is available and what is needed, and rehabilitation. Both require support from analysts who are familiar with both the legacy system and the new system. For software assets, rehabilitation usually requires the support of the new system's architect, who will direct how the assets will be integrated into the new architecture.

For software assets, focus first on large-grained assets that can be wrapped or that will require only interface changes rather than changes in large chunks of the underlying algorithms. Determine how the candidate asset can fit into the architecture of the targeted new system. Don't forget to consider the requirements for performance, modifiability, reliability, and other nonbehavioral qualities. Also, don't forget to include all the nonsoftware assets associated with the software: requirements, design, test, and management artifacts.

Once the existing assets have been organized and understood and candidate assets for mining have been identified, the rehabilitation of these assets can begin. In many ways, a mining initiative that involves extensive rehabilitation of assets can resemble a reengineering project [Seacord 03, Sneed 01, Ulrich 02] or a development project in its own right. Technical planning (as in the "Technical Planning" practice area) can help in planning and coordinating the effort.

Aspects Peculiar to Product Lines

Mined assets for a product line must have the same qualities as newly developed core assets. Mined assets must be (re)packaged with reuse in mind, must meet the product line requirements, must align with the product line architecture, and must meet the quality goals consistent with the goals of the product line. Product lines must focus on the strategic, large-grained reuse of the mined assets. The primary issues that motivate large-scale reuse for a product line are schedule, cost, and quality. The mined and rehabilitated assets must meet the needs of the plurality of systems in the product line. A product line accommodates a longer and wider view of future system change; any mined asset must be robust enough to accommodate such change gracefully.

When mining an asset (software or otherwise) for a software product line, consider:

- its alignment with requirements for immediate products in terms of both common features and variation points
- its appropriateness for potential future products

- the amount of effort required to make the asset's interface conform to the constraints of the product line architecture
- the extensibility of the asset with respect to its potential future based on the future evolution that will be required of the architecture
- its maintenance history
- other assets (for example, script and data files) that may be required from the legacy system
- projected long term cost of the mined asset

When mining software assets for single systems, we look for components that perform specific functions well. However, for product line systems, quality attributes such as maintainability and suitability become more important over time. Thus, we might accept mined assets for product lines that are suboptimal in fulfilling specific tasks if they meet the critical quality-attribute goals. An asset's total cost of ownership across the products for which it will be used should be lower than the sum of similar assets mined for one-time use.

Application to Core Asset Development

The process of mining existing assets is largely about finding suitable candidates for core assets of the product line. Software assets that are well structured and well documented and have been used effectively over long periods of time can sometimes be included as product line core assets with little or no change. Software assets that can be wrapped to satisfy new interoperability requirements are also desirable. On the other hand, assets that don't satisfy these requirements are undesirable and may have higher maintenance costs over the long term. Depending on the legacy inventory and its quality, an assortment of candidate assets is possible, from architectures to small pieces of code.

An existing architecture should be analyzed carefully before being accepted as the pivotal core asset—the product line architecture. See the "Architecture Evaluation" practice area for a discussion of what that analysis should entail.

Candidate software assets must align with the product line architecture, meet specified component behavior requirements, and accommodate any specified variation points. In some cases, a mined component may represent a potentially valuable core asset but won't fit directly into the product line architecture. Usually, the component will need to be changed to accommodate the constraints of the architecture. Sometimes a change in the architecture might be easier, but of course this will have implications for other components, for the satisfaction of quality goals, and for the support of the products in the product line.

Once in the product line core asset base, mined assets are treated in the same way as newly developed assets.

Application to Product Development

It is possible and reasonable to use mined assets for components that are unique to a single product in the product line, but in this case the mining activity will become indistinguishable from mining in the non-product-line case. The same issues discussed above (paying attention to quality attributes, architecture, cost, and time-to-market) will still apply. And it will be worth taking a long, hard look at whether the mined component really is unique to a single product or could be used in other products as well, thus making the cost of its rehabilitation more palatable. In that case, the team responsible for mining would be wise to look for places where variability could be installed in the future, should the asset in question ever turn out to be useful in a group of products.

Specific Practices

Options Analysis for Reengineering (OAR): OAR is a method that can be used to evaluate the feasibility and economy of mining existing components for a product line. OAR operates like a funnel in which a large set of potential assets is screened out so that the effort can most efficiently focus on a smaller set that will most effectively meet the technical and programmatic needs of the product line. OAR prescribes the following steps [Bergey 01, Bergey 02a, Bergey03].

1. **Establish mining context:** First, capture your organization's product line approach, legacy base, and expectations for mining components. Establish the programmatic and technical drivers for the effort, catalogue the documentation available from the legacy systems, and identify a broad set of candidate components for mining. This task establishes the needs of the mining effort and begins to illuminate the types of assets that will be most relevant for mining. It also identifies the documentation and artifacts that are available, and it enables focused efforts to close gaps in existing documentation.
2. **Inventory components:** Next, identify the legacy system components that can potentially be mined for use in a product line core asset base. During this activity, identify required characteristics of the components (such as functionality, language, infrastructure support, and interfaces) in the context of the product line architecture. This activity creates an inventory of candidate legacy components together with a list of the relevant characteristics of those components. It also creates a list of those needs that cannot be satisfied through the mining effort.
3. **Analyze candidate components:** Next, analyze the candidate set of legacy components in more detail to evaluate their potential for use as product line components. Screen them on the basis of how well they match the required characteristics. This activity provides a list of candidate components, together with estimates of the cost and effort required for rehabilitating those components.
4. **Analyze mining options:** Next, analyze the feasibility and viability of mining various aggregations of components on the basis of cost, effort, and risk. Assemble different aggregations of components and weigh their costs, benefits, and risks.

5. **Select mining option:** Finally, select the mining option that can best satisfy the organization's mining goals by balancing the programmatic and technical considerations. First, establish drivers for making a final decision, such as cost, schedule, risks and difficulty. Tradeoffs often can be established by this activity. Evaluate each mining option (component aggregation) on the basis of how well it satisfies the most critical driver. Select an option, and then develop a final report to communicate the results.

OAR has been used to make decisions on mining components for a satellite tracking system [Bergey 01]. OAR has also been used to evaluate the extent to which components proposed by suppliers for reuse in a product line meet the product line's stated needs. It has evaluated the types of changes required to fit the component into the product line [Bergey 03, Muller 03]. OAR is in the process of being extended to handle other asset types such as unit test cases and documentation.

Architecture recovery/reconstruction tools: Some tools that are available to assist in the architecture reconstruction process include Rigi [Muller 88], the Software Bookshelf [Finnegan 97], DISCOVER [Tilley 98], and the Dali workbench [Kazman 98] and the ARMIN tool [O'Brien 03].

The ARMIN tool is a flexible, lightweight tool for architecture reconstruction. Other tools are used to extract information that is then used by ARMIN to generate architectural views. Using ARMIN involves five steps:

1. Information extraction, the activity uses tools such as parsers to extract information from existing design and implementation artifacts such as the source code.
2. Database construction, which stores the extracted information in a database for future analysis. This may involve changing the format of the data.
3. View fusion, which augments the extracted information by combining information to generate a set of low-level views of the software.
4. Architecture view composition, which generates a set of architecture views through abstraction and visualizes these views and enables the user to explore and manipulate views.
5. 5. Architecture analysis, which evaluates the resultant architecture and in some cases evaluates the conformance of the as-built architecture obtained from reconstruction to an as-designed architecture.

Tool support makes mining undocumented software assets more effective and significantly less cumbersome by reducing the time it takes to ascertain what a piece of software does and how it interacts with other parts of the system. Tools can be brought to bear that automatically chart interconnections of various kinds among software elements. More valuable than tools, however, are the people who worked on and are knowledgeable about the legacy software. Find them if you can. They can tell you the strengths and weaknesses of the software that weren't written down, and they can give you the "inside story" that no tool can hope to recover.

Mining Architectures: In some cases the software architecture of an existing system can become the product line architecture. Mining Architectures for Product Lines (MAP) is a method that determines whether the architectures of existing systems are similar and whether the corresponding systems have the potential of becoming a software product line [O'Brien 01]. The MAP method combines techniques for architecture reconstruction and product line analysis to analyze the architectural patterns and attributes of a set of systems. This analysis determines if there are similar components and connections between the components within these systems and examines their commonalities and variabilities. MAP has been used in the development of a prototype product line architecture for a sunroof system. MAP and OAR can also be used together where MAP supports decision-making on reusing architectures, while OAR supports decision-making on identifying components that fit within the constraints of the architecture.

Requirements Reuse and Feature Interaction Management: Developers realize that complex applications are often best built by using a number of different components, each performing a specialized set of services. But the components, each embodying different requirements in different service domains, can interact in unpredictable ways. How to design components to minimize or at least manage interaction is a current issue. This problem of interaction becomes even more significant when reusing requirements. Interactions must be detected and resolved in the absence of a specific implementation framework. Shehata et al. stresses that an understanding of interaction management is key to understanding how to reuse requirements and describes a conceptual process framework for formulating and reusing requirements [Shehata 02]. Reusable requirements are classified into three different levels of abstraction for software requirements: domain-specific requirements, generic requirements and domain-requirements frameworks. This classification is used as the basis for a reusability plan to support the view of the importance of interaction management.

Wrapping: Wrapping involves changing the interface of a component to comply with a new architecture, but not making other changes in the component's internals. In fact, pure wrapping involves no change whatsoever in the component, but only interposing a new thin layer of software between the original component and its clients. That thin layer provides the new interface by translating to and from the old. There are enormous advantages to reusing existing assets with little or no internal modification through wrapping. As soon as any modification takes place, the associated documentation changes, the test cases change, and a ripple effect takes place that influences other associated software. Wrapping prevents this and allows the "as-is" reuse of many of the assets associated with the software component, such as its test cases and internal design documentation. The idea is to translate the "as-is" interface to the "to-be" interface. Weiderman et al. discuss some of the available wrapping techniques [Weiderman 97]. Seacord [Seacord 01] discusses a case study that applied several wrapping techniques.

Adapting components: Software components that are being used in a context other than the one for which they were originally developed often do not exactly fit their assigned roles. There are a couple of techniques for accommodating these differences. The adapter design pattern [Gamma 95] imposes an intermediary between two components. The adapter can compensate for mismatches in number or types of parameters within a service signature, provide synchronization in a multithreaded interaction, and adjust for many other types of incompatibilities. Scripting languages can often be used to implement the adapter.

Practice Risks

The major risks associated with mining are (1) failure to find the right assets and (2) choosing the wrong assets. Both will result in schedule slippage and opportunity cost in terms of what other productive activities the staff could have been carrying out. A secondary risk is inadequate support for the mining operation, which will result in a failed operation and the (misguided) impression that mining is not a viable option.

Specific risks associated with an unsuccessful search operation include:

- **Flawed search:** The search for reusable assets may be fruitless, resulting in a waste of time and resources. Or, relevant assets may be overlooked, resulting in time and resources being wasted duplication of what already exists. A special case of the latter is when noncode assets are shortsightedly ignored. To minimize both of these risks, build a catalogue of your reusable assets (including noncode assets) and treat that catalogue as a core asset of the product line. It will save time and effort next time.
- **Overly successful search:** There may be too many similar assets, resulting in too much effort spent on analysis.
- **Fuzzy criteria:** The criteria for what to search for need to be crisp enough so that an overly successful search is avoided, yet general enough so that not all viable candidates are ruled out.
- **Failure to search for nonsoftware assets:** Failure to consider nonsoftware assets in your search, such as specifications, test suites, procedures, budgets, work plans, requirements, and design rationale, will reduce the effectiveness of any mining operation.
- **Inappropriate assets:** Assets recovered from a search may appear to be usable but later turn out to be of inferior quality or unable to accommodate the scope of variation required.
- **Bad rehabilitation estimates:** Initial estimates of the cost of rehabilitation may be inadequate, leading to escalating and unpredictable costs.

Organizational issues leading to mining risks include:

- **Lack of corporate memory:** Corporate memory may not be able to provide sufficient data to utilize the software asset effectively.
- **Inappropriate methods:** The wrong reengineering methods and tools may be selected, leading to schedule and cost overruns.
- **Lack of tools:** Tools required for the mining effort may not be integrated to the extent necessary, leading to risky and expensive workarounds.

- **Turf conflicts:** Potential turf conflicts may undermine the decision process in selecting between similar candidate assets. Or, a repository of assets may be off limits for political or organizational reasons.
- **Inability to tap needed resources:** There may be an inability to free resources from the group that originally created the component to rehabilitate or renovate it.

Further Reading

[Seacord 03]: This book on modernizing legacy systems by Seacord et al. provides guidance on how to implement a successful modernization strategy and specifically describes a risk-managed, incremental approach that encompasses changes in software technologies, engineering processes, and business practices.

Software System Integration

Software system integration refers to the practice of combining individual software components into an integrated whole. Software is integrated when components are combined into subsystems or when subsystems are combined into products. In a waterfall model, software system integration appears as a discrete step toward the end of the development life cycle between component development and integration testing. In an incremental model, integration is an ongoing activity; components and subsystems are integrated as they are developed into multiple working mini-versions of the system. An incremental approach to integration decreases risk, because problems encountered during software integration are often the most complex. Object technologists are proponents of incremental development, and object-oriented development methods are based on the principle of ongoing integration practices

Integration is bound up in the concept of component interfaces. Recall from the "Architecture Definition" practice area that an interface between two components is the set of assumptions that the programmers of each component can safely make about the other component [Parnas 72]. This includes its behavior, the resources it consumes, how it acts in the face of an error, and other assumptions that should be documented as part of a component's interface. This definition is in stark contrast to the simplistic (and quite insufficient) notion of "interface" that merely refers to the "signature" or syntactic interface that includes only the program's names and parameter types. This definition of "interface" may let two components compile together successfully, but only the Parnas definition (which subsumes the simpler one) will let two components work together correctly. When interfaces are defined thoughtfully and documented carefully, integration proceeds much more smoothly because the interfaces define how the components will connect to and work with each other.

Aspects Peculiar To Product Lines

In a product line effort, software system integration occurs during the installation of core assets into the core asset base and also during the building of an individual product. In the former case, preintegrating as many of the software core assets as you can will make product-building a much more economical operation [Clements 01a, p. 118]. In either case, you need to consider integration early on in the development of the production plan and architecture for the entire product line. The goal is to make software system integration more straightforward and predictable.

In a product line, the effort involved in software system integration lies along a spectrum. At one end, the effort is almost zero. If you know all of the products' variabilities in advance, you can produce an integrated parameterized template of a generic system with formal parameters. You can then generate final products by supplying the actual parameters specific to the individual product requirements and then launching the construction tool (along the lines of the Unix "make" utility). In this case, each product consists entirely of core components; no product-specific code exists. This is the "system generation" end of the integration spectrum.

At the other end of the spectrum, considerable coding may be involved to bring together the right core components into a cohesive whole. Perhaps the components need to be wrapped, or perhaps new components need to be designed and implemented especially for the product. In this case, the integration more closely resembles that of a single-system project.

Most software product lines occupy a middle point on the spectrum. Obviously, the closer to the generation side of the spectrum you can align your production approach, the easier integration will be and the more products you will be able to turn out in a short period of time. For example, it used to take Cummins Inc. about a year to bring new engine-control software to the point of acceptance testing. Now, after adopting a product line approach, they can do it in about a week [Clements 01a, p. 417-442]

However, circumstances may prevent you from achieving pure generation. Perhaps a new product has features you have not considered. Perhaps your application area prevents you from knowing all of the variabilities up front. Or perhaps the variabilities are so numerous or complex or interact with each other in such complicated ways, that building the construction tool will be too expensive. And it may be that you do not want to turn out many products in a short amount of time, but fewer products spread out over even periods. In that case, the construction tool may be less appealing.

In software system integration for product lines, the cost of integration is amortized over many products. Once the product line scope, core assets, and production plan have been established in the core asset base, and a few systems have been produced from that base, most of the software system integration work has been done. The interfaces have been defined, and they work predictably. They have been tested. Components work with one another. In subsequent variations and adaptations of the product, there is relatively little software system integration effort when the variations and adaptations occur within components. Even when new components are being added with new interfaces, the models from previous interfaces can and should be followed, thus minimizing the work and the risk of integration. So, in a very real sense, products (after the first one or two) tend to be "preintegrated" such that there are few surprises when a system comes together.

Application to Core Asset Development

When core assets are developed, acquired, or mined, remember to take integration into account. Try to specify component interfaces not solely in natural language but in machine-checkable form. Using languages such as Interface Description Language (IDL), the syntactic or "signature" part of the interfaces can be specified early and kept current continuously throughout the development process. Early on, the bodies for these specifications can be stubbed out so that the code can be compiled and checked by a machine for consistency. Absence of consistency errors does not guarantee smooth integration—the components might assemble smoothly but still fail to work correctly together—but it's a good start.

Evaluate any components you mine or acquire for their integrability and their granularity. A component is "integrable" if its interfaces (in the Parnas sense) are well defined and well documented so that it can potentially be wrapped for commonality with other components (if not used with assurance as is). Finally, remember that it is generally easier to build a system from small numbers of large, preintegrated pieces than from large numbers of small, unintegrated components.

Application to Product Development

A big benefit of product line practice is that software system integration costs tend to decrease for each of the subsequent products in the product line. If the production plan calls for the addition of components or internal changes in components, some integration may be required depending on the nature of the changes. Finally, in the system generation case, integration becomes a matter of providing values for the parameters and launching the construction tool. The key in all of these cases is that the integration occurs according to a preordained and tested scheme.

Specific Practices

Interface languages: Programming languages such as IDL allow you to define machine-independent syntactic interfaces. Programming languages such as Ada allow you to define a compilable specification separate from the body. Ada programmers have found that keeping a continuously integrated system using full specifications and stubbed bodies decreases the integration time and costs dramatically. These languages and others do not allow specification of the full semantic interfaces of components, but catching signature-level integration bugs early is a win.

This practice applies primarily to the development of new components but retains the leverage for subsequent products in a product line. One of the principal aspects of CelsiusTech's product line solution was the institutionalization of continuous integration using Ada rather than the more traditional all-at-once approach [Brownsword 96]. Most object-oriented design techniques prescribe the development of architectural frameworks and the use of patterns; both have been proven to support product lines and facilitate software integration.

Wrapping: Wrapping, described as a specific practice in the "Mining Existing Assets" practice area, involves writing a small piece of software to mediate between the interface that a component user expects and the interface that the used component comes with. Wrapping is a technique for integrating components whose interfaces you do not control, such as components that you have mined or acquired from a third party [Seacord 01].

Middleware: An especially integrable kind of architecture employs a specific class of software products to be the intermediaries between user interfaces on the one hand and the data generators and repositories on the other. Such software is called "middleware" and is used in connection with Distributed Object Technology (DOT) [Wallnau 97]. There are three prominent examples of middleware standards and technology. One is the Common Object Request Broker Architecture (CORBA) and its various commercial implementations [OMG 96]. The second is the Distributed Component Object Model (DCOM). The third is the proprietary middleware solution that has grown around the Java programming language. Middleware is discussed in more detail in the "Architecture Definition" practice area.

System generation: In some limited cases, a new product in a product line can be produced with no software system integration at all. These are cases in which all (or most) of the product line variability is known in advance. In these cases, it may be possible to have a template system from which a computer program produces the new products in the product line simply by specifying variabilities as actual parameters. Such a program is called a "system generator." One example of such a family of products would be an operating system in which all of the variabilities of the system are known ahead of time. Then, to generate the operating system, the "sysgen" program is simply provided with a list of system parameters (such as, processor, disk, and peripheral types, and their performance characteristics), and the program produces a tailored operating system rather than integrating all the components of an operating system.

FAST generators: In *Software Product-Line Engineering* [Weiss 99], Weiss and Lai describe a process for building families of systems using generator technology. The Family-Oriented Abstraction, Specification, and Translation (FAST) process begins by explicitly identifying specific commonalities and variabilities among potential family members and then designing a small special-purpose language to express both. The language is used as the basis for building a generator. Turning out a new family member (product) is then simply a matter of describing the product in the language and "compiling" that description to produce the product.

Practice Risks

The major risks associated with software system integration include:

- **Natural-language interface documentation:** Relying too heavily on natural language for system interface documentation and not relying heavily enough on the automated checking of system interfaces will lead to integration errors. Natural language interfaces are imprecise, incomplete, and error-prone. Carrying forward in the face of undetected interface errors increases the cost of correcting such errors and increases the overall cost of integration. Automated tools, however, are more oriented to syntactic checking and are less effective at checking race conditions, semantic mismatch, fidelity mismatch, and so on. Some interface specifications must still be done largely with natural language and are still error-prone.
- **Component granularity:** There is a risk in trying to integrate components that are too small. The cost of integration is directly proportional to the number and size of the interfaces. If the components are small, the number of interfaces increases proportionally, if not geometrically, depending on the connections they have to each other. This leads to greatly increased testing time. One of the lessons of the CelsiusTech case study was that "CelsiusTech found it economically infeasible to integrate large systems at the Ada-unit level" [Brownsword 96]. Although the component granularity is dictated by the architecture, we capture the risk here, because this is where the consequence will make itself known.
- **Variation support:** There is a risk in trying to make variations and adaptations that are too large or too different from existing components. When new components or subsystems are added, they must be integrated. Variations and adaptations within components are relatively inexpensive as far as system integration is concerned, but new components may cause architectural changes that structure the product in ways that cause integration problems.

Further Reading

[Weiss 99] describes the Family-Oriented Abstraction, Specification, and Translation (FAST) process, which includes a generator-building step that essentially obviates the integration phase of product development.

[Wallnau 97] provides a nicely digestible overview of middleware.

---

1. Our use of the word framework is meant to suggest a conceptual index, a frame of reference, for the information essential to success with software product lines. We are using the dictionary definition with no intended connections to current technical usages in the vein of architectural frameworks or application frameworks.