1

Basic Ideas and Terms

We trumpeted software product lines in our introduction to Part I, raving about the benefits that were possible and hinting about some potential risks. In this chapter we will go beneath the surface and examine the associated ideas and terms more closely. We begin by answering the question: What is a software product line?

1.1 What Is a Software Product Line?

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 definition is consistent with the definition traditionally given for any product line—a set of systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission. 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 software product lines 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. If you look carefully, you will find plenty of examples all around you.

Product Lines Everywhere

In his bestseller, *Chaos: Making a New Science*, James Gleick relates how some of the pioneers of chaos theory would, while relaxing in their favorite coffeehouse, compete to find the nearest example of a certain kind of chaotic system [Gleick 87, p. 262]. A flag whipping in the breeze, a dripping faucet, a rattling car fender—they seemed to be everywhere.

I can relate. Lately it seems that no matter where I turn I see a product line. At airports I see product lines of airliners (such as the Airbus A-318, A-319, A-320, and A-321, a family of aircraft that range from 100 to 220 seats but clearly share production commonalities) powered by product lines of jet engines and equipped with product lines of navigation and communication equipment. When I arrive at my destination, I rent an American midsize car that is always pretty much the same except for cosmetic factors and features, even though it could have any one of four nameplates on it. I wonder how much more expensive the cars would be if they had nothing in common. The hotel leaves a copy of the local newspaper at my door: the morning edition of the citywide version. Someone else will get the afternoon edition of the upstate version, but it will have most of the same stories, will have all of the same comics, and will come off the same presses. On my way to work I pass residential subdivisions where the houses are all variants on a few basic designs. Even the street signs are the same except for the names of the streets. While the actual street name is fundamental to a street sign's function, it is inconsequential to its fabrication and is just a variation point.

We know that product lines have been around in manufacturing almost since manufacturing began. Remember Eli Whitney's idea of interchangeable parts for rifles in the early 1800s? This idea made it possible to build a product line of firearms that shared components. Remember the IBM System/360 family of computers? From the Principles of Operation:

> *Models of System/360 differ in storage speed, storage width (the amount of data obtained in each storage access), register width, and*

> *capabilities for processing data concurrently with the operation of multiple input/output devices. Several CPU's permit a wide choice in internal performance. Yet none of these differences affects the logical appearance of these models to the programmer. An individual System/360 is obtained by selecting the system components most suited to the applications from a wide variety of alternatives in internal performance, functional ability, and input/output (I/O).*

This was clearly a product line, and the operating system that powered it was a software product line. And town plans in which the buildings look like each other predate post-War suburbia by at least eight centuries. During the Pei Sung dynasty of northern China (960–1127 A.D.), a book called the "Ying-tsao fa-shih" was written by Li Chieh, the state architect of the emperor Hui-tsung, in 1100 A.D. and published in 1103 A.D. This was a set of building codes for official buildings. It described in encyclopedic detail the layouts, materials, and practices for designing and building official buildings. It listed standard parts and standard ways of connecting the parts as well as recognizing and parameterizing variations of the parts such as allowable lengths, load capacities, bracketing, decorations, allowed components based on the building's purpose, and the options available for various component choices. The book also included design construction details that provided a process for building design and implementation of that design. While it was influential in spreading the most advanced techniques of the time of its first publication in 1103, by codifying practice it may also have inhibited further development and contributed to the conservatism of later techniques. Some scholars even claim that because of it, Chinese architecture remained largely unchanged until the beginning of the twentieth century. (In a product line, you've got to know when your architecture has outlived its usefulness.)

However, like Gleick's scientists, I find some of the best examples of product lines in places where I go to eat. Here's something I saw on the menu at a little Mexican restaurant recently:

> *#16. Enchiladas verdes: Corn tortillas baked with a zesty filling, covered with a green tomatillo sauce. Your choice of chicken, beef, pork, or cheese.*
>
> *#17. Enchiladas rojas: Corn tortillas baked with a zesty filling, covered with a red ancho chile sauce. Your choice of chicken, beef, or pork.*

See what I mean? This restaurant clearly produces an "enchilada" product line. (Well, all right, "clearly" applies only to those of us who have been thinking about this for too long.) While admittedly a cheesy example (sorry), it actually provides a pretty good analogy with *software* product lines and the central concepts they embody.

The enchilada product line consists of seven separate products, differentiated by filling and sauce. This defines their variabilities. The corn tortillas are core assets because they're used in every product. The red and green sauces are also core assets because they're used in four and three products, respectively. And the meat fillings are also core assets, used in two products each. But the cheese is a product-specific asset, used only in the enchiladas verdes.

Some of the core assets have attached processes that indicate how they are to be instantiated for use in products. Here, the beef, pork, and chicken have attached processes that dictate how they're chopped, seasoned, and cooked. The processes call for different spices to be added depending on the sauce.

All of the products share an "architecture"—tortillas wrapped around a filling, covered with sauce. And they also share a "production plan": prepare filling, wrap filling in tortilla, cover with sauce, bake at 350 degrees for 15 minutes, garnish, serve.

This little product line provides economies of scope; the common ingredients let the restaurant stock a small number of food items delivered from a small number of suppliers. They provide personnel flexibility: the same person who makes the pork enchiladas rojas is, I would bet my house, the same person who makes the cheese enchiladas verdes. And because the choices are limited, many of the ingredients can be pre-prepared, allowing for rapid time-to-market, which in this case means time-to-table.

As a family, the products define a clear scope that leaves little doubt what's in and what's out. Chicken enchiladas are in. Beef enchiladas are in. And if you wanted cheese enchiladas with the red sauce instead of the green, well, that's probably open for discussion. As we'll see, a scope definition with a pronounced gray area is a healthy thing—but duck enchiladas with a white sauce are definitely out.

Finally, because the commonalities and variabilities are exquisitely clear, it's easy to see how this product line's scope could be expanded, by offering new fillings and new sauces and perhaps new combinations. You could even see how this efficient production capability could be used to launch an entirely new product line to capture a new market segment: replace the corn tortillas with flour tortillas, lose the sauce, add lettuce and tomato and other condiments, and open a new restaurant chain that sells "wraps."

If you already had a strong grasp of the concepts underlying software product lines, this little culinary diversion probably had no effect on you, except possibly to make you hungry. If you didn't, perhaps the concepts are now a bit more palpable for you. In either case, the next time you're at a coffeehouse or restaurant, try looking around to see how many product lines you can spot.

¡Buen provecho!

—PCC

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 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.

Other Voices: Beyond Technology

The primary thesis of this book is that software product lines, although enmeshed in the highly technological field of software, rely on much more than technology to succeed. Martin Griss, a reuse expert at Hewlett-Packard and co-author of the highly regarded *Software Reuse: Architecture, Process, and Organization for Business Success* [Jacbobson 97], put it this way [Griss 95]:

> *. . . In almost all cases, a simple architecture, a separate component group, a stable application domain, standards, and organizational support are the keys to success. Correct handling of these (largely non-technical) issues is almost always more critical to successful reuse than the choice of specific language or design method, yet too many . . . experts choose to ignore these factors.*
>
> *. . . Over the last 10 years, software reuse researchers and practitioners have learned that success with systematic reuse requires careful attention be paid to both technical and nontechnical issues. Furthermore, the nontechnical issues are more pervasive and complex than was realized at first. Without a systematic and joint focus on people, process, and product issues, a project will not succeed at managing the scope and magnitude of the changes and investment necessary to achieve reuse. Simply creating and announcing a reusable class library will not work. Without a "reuse mindset," organizational support, and methodical processes directed at the design and construction of appropriate reusable assets, the reuse investment will not be worthwhile.*
>
> *. . . Often, sweeping changes in the software development organization are needed to institute large-scale, systematic reuse. These include business, process, management, and organizational changes . . . .*

Software product lines give you *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.

Other Voices: Japanese Software Factories and Economies of Scope

In 1991, Michael Cusumano's *Japan's Software Factories* [Cusumano 91] burst onto the scene, revealing the "secrets" of the Japanese software companies that, at the time, it was feared were going to bury American and European firms with their sky-high productivity rates. And what was their secret? Largely it was what we would call today a software product line approach, based on that feeling of déjà vu that all eventual product line managers share: "You know, I could swear we've built this product already." The driving concept of economies of scope was born. Cusumano wrote:

> *The Japanese software facilities discussed in this book differed in some respects, reflecting variations in products, competitive strategies, organizational structures, and management styles. Nonetheless, the approaches of Hitachi, Toshiba, NEC, and Fujitsu had far more elements in common than in contrast, as each firm attempted the* strategic management and integration *of activities required in software production, as well as the achievement of* planned economies of scope—*cost reductions or productivity gains that come from developing a series of products within one firm (or facility) more efficiently than building each product from scratch in a separate project. Planned scope economies thus required the deliberate (rather than accidental) sharing of resources across different projects, such as product specifications and designs, executable code, tools, methods, documentation and manuals, test cases, and personnel experience. It appears that scope economies helped firms combine process efficiency with flexibility, allowing them to deliver seemingly unique or tailored products with higher levels of productivity than if they had not shared resources among multiple projects.*
>
> *Japanese managers did not adopt factory models and pursue scope economies simply out of faith. Detailed studies concluded that as much as 90 percent of the programs they developed in any given year, especially in business applications, appeared similar to work they had done in the past, with designs of product components falling into a limited number of patterns. Such observations convinced managers of the possibility for greater efficiencies, in scope if not in scale, and set an agenda for process improvement. Companies subsequently established facilities focused on similar products, collected productivity and quality data, standardized tools and techniques, and instituted appropriate goals and controls. As the factory discussions demonstrate, Japanese firms managed in this way not simply one or two special projects for a few years. They established permanent software facilities and research*

and development efforts, as well as emphasized several common elements in managing across a series of projects [not the least of which was a commitment to process improvement].

1.2 What Software Product Lines Are Not

There are many approaches that at first blush could be confused with software product lines. In fact, you might be asking: "Isn't software product line just a new name for *x*?" Though we certainly want you to build on previous knowledge and experience, we want to ensure from the outset that you don't erroneously equate software product lines with something they are not. Describing what you don't mean is often as instructive as describing what do you mean. When we speak of software product lines, we don't mean any of the following:

1.2.1 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.

1.2.2 Single-System Development with Reuse

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. 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. You need to maintain two systems as entirely separate entities. 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 core assets. 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.

1.2.3 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 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 alone also lacks the technical and organizational management aspects that are so important to the success of a software product.

1.2.4 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 asset base.

1.2.5 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. 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.

1.2.6 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 at architecture and design reviews to ensure that the standards are being followed. For example, the developer might be able to select between two 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 inputted to the software product line, no more.

1.3 A Note on Terminology

Now that we have covered what we don't mean, the following terms lay out what we do mean:

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 Section 1.1.

Core assets are those assets 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 the organization). 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 provides 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*. Figure 1.1 shows the mapping between our terminology and this different set of terms.

To us, the terminology is not as important as the concepts. That having been said, you might encounter both sets of terms in other places and should be able to translate between them. You might also use an entirely different set of terms that are equivalent to the ones we use. In that case you will probably want to do your own mapping, akin to that shown in Figure 1.1, before you proceed with the rest of the book. Although we have tried not to invent vocabulary, as you read on you may find other terms we use that you may call by different names, and you may want to expand your map.

The next chapter discusses the benefits (and the risks) of software product lines.

Our Terminology		Alternate Terminology
Product Line	→	Product Family
Core Assets	→	Platform
Business Unit	→	Product Line

Figure 1.1 Alternate Terminology

1.4 For Further Reading

See Section 3.6.

1.5 Discussion Questions

1. Some would argue that a software product line is just the group of products produced by a single business unit (profit/loss center). What distinguishes this definition from the one we gave in Section 1.3?
2. What is the difference between software product line practice and domain engineering? What is the difference between software product line practice and application engineering?
3. Suppose your organization has a library of components *and* a reconfigurable architecture to support a family of products that have many common features. Do you have a software product line? If not, what is missing?
4. Describe an experience you have had with software reuse. Identify the similarities and differences between software product line practice and your experience.