Chapter 15
Software Product Lines

Mark Staples

15.1 Product Lines for ICDE

The ICDE system is a platform for capturing and disseminating information that can be used in different application domains. However, like any generically applicable horizontal technology, its broad appeal is both a strength and weakness. The weakness stems from the fact that a user organization will need to tailor the technology to suit its application domain (e.g., finance), and make it easy for their users to learn and exploit. This takes time and money, and is hence a disincentive to adoption.

Recognizing this, the product development team decided to produce a tailored version of the ICDE platform for their three major application domains, namely financial analysis, intelligence analysis and government policy research. Each of the three would be marketed as different products, and contain specific components that make the base ICDE platform more user-friendly in the targeted application domain.

To achieve this, the team brainstormed several strategies that they could employ to minimize the design and development effort of the three different products. The basic idea they settled on was to use the base ICDE platform unchanged in each of the three products. They would then create additional domain-specific components on top of the base platform, and build the resulting products by compiling the base platform with the domain-specific components. This basic architecture is depicted in Fig. 15.1.

What the team had done was to take the first steps to creating a product line architecture for their ICDE technology. Product lines are a way of structuring and managing the on-going development of a collection of related products in a highly efficient and cost-effective manner. Product lines achieve significant cost and effort reductions through large scale reuse of software product assets such as architectures, components, test cases and documentation.

The ICDE product development team already benefits from software reuse in a few different ways. They reuse some generic libraries (like JDBC drivers to handle database access), and entire off the shelf applications (like the relational database in

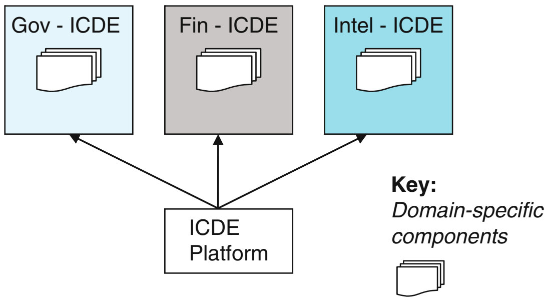

Fig. 15.1 Developing domain-specific products for the ICDE platform

the ICDE data store). Market forces are driving the introduction of the three tailored versions of the ICDE product. But if the team developed each of these separately, it could triple their development or maintenance workload. Hence their plan is to reuse core components for the fundamental ICDE functionality and to create custom components for the functionality specific to each of the three product's markets. This is a kind of software product line development, and it should significantly reduce their development and maintenance costs.

The remainder of this chapter overviews product line development and architectures, and describes a range of reuse and variation mechanisms that can be adopted for product line development.

15.2 Software Product Lines

Widespread software reuse is a "holy grail" for software engineering. It promises a harmonious world where developers can quickly assemble high-quality solutions from a suite of preexisting software components. The quest for effective software reuse has in the past stereotypically focused on "reuse in the small," exploiting techniques to reuse individual functions, or libraries of functions for data-types and domain-independent technologies. Collection class and mathematical function libraries are good examples. Such approaches are proven to be beneficial, but they have not realized the full promise of software reuse.

Reusing software is easy if you know it already does exactly what you want. But software that does "almost" what you want is usually completely useless. For this reason, to realize the full benefits of software reuse, we need to practice effective "software variation" as well. Modern approaches to software reuse, such as Software Product Line (SPL) development, support software variation "in the large," with an architectural basis and a domain-specific focus. Software Product Line (SPL) development has proven to be an effective way to benefit from software reuse and variation. It has allowed many organizations to reduce development costs, reduce development duration, and increase product quality.

In SPL development, a collection of related products is developed by combining reused core assets with product-specific custom assets that vary the functionality

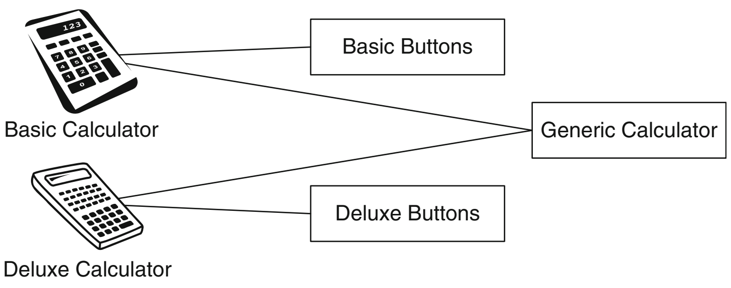

Fig. 15.2 A schematic view of a simple product line

provided by the core assets. A simple conceptual example of a product line is shown in Fig. 15.2. In the picture, two different calculator products are developed, with both using the same core asset internal boards. The different functionalities of the two calculator products are made available by each of their custom assets, including the two different kinds of buttons that provide the individualized interface to the generic, reused functionality.

From this simple perspective, SPL development is just like more traditional hardware-based product line development, except that in SPL development, the products are of course software![1]

For any product in a SPL, almost everything is implemented by reused core assets. These core assets implement base functionality which is uniform across products in the SPL, as well as providing support for variable features which can be selected by individual products. Core asset variation points provide an interface to select from among this variable functionality. Product-specific custom assets instantiate the core assets' variation points, and may also implement entire product-specific features.

Software variation has a number of roles in SPL development. The most obvious role is to support functional differences in the features of the SPL. Software variation can also be used to support nonfunctional differences (such as performance, scalability, or security) in features of the SPL.

SPL development is not simply a matter of architecture, design, and programming. SPL development impacts existing processes across the software development lifecycle, and requires new dimensions of process capability for the management of reused assets, products, and the overarching SPL itself. The Software Engineering Institute has published Product Line Practice guidelines (see Further Reading at the end of the chapter) for these processes and activities that support SPL development. We will refer to these practice areas later within this chapter.

[1]Product lines are also widely used in the embedded systems domain, where products are a software/hardware combination.

15.2.1 Benefiting from SPL Development

When an organization develops a set of products that share many commonalities, a SPL becomes a good approach. Typically an organization's SPL addresses a broad market area, and each product in the SPL targets a specific market segment. Some organizations also use an SPL to develop and maintain variants of a standard product for each of their individual customers.

The scope of a product line is the range of possible variations supported by the core assets in a SPL. The actual products in a SPL will normally be within the SPL scope, but custom assets provide the possibility for developing functionality beyond the normal scope of the SPL. To maximize the benefit from SPL development, the SPL scope should closely match both the markets of interest to the company (to allow new products within those markets to be developed quickly and efficiently), and also the full range of functionality required by the actual products developed by the company. These three different categories of product (the company's markets of interest, the SPL scope, and the actual products developed by the company) are depicted in a Venn diagram in Fig. 15.3.

The most obvious benefit from SPL development is increased productivity. The costs of developing and maintaining core assets are not borne by each product separately, but are instead spread across all products in the SPL. Organizations can capture these economies of scale to benefit from the development of large numbers of products. The SPL approach scales well with growth, as the marginal cost of adding a new product should be small.

However, SPL development also has other significant benefits. When the core assets in an SPL are well established, the time required to create a new product in the SPL is much smaller than with traditional development. Instead of having to wait for the redevelopment of functionality in the core assets, customers need only wait for the development of functionality that is unique to their needs.

Organizations can also experience product quality benefits from SPL development. In traditional product development, a defect might be repeated across many products, but in SPL development, a defect in a core asset only needs to be fixed

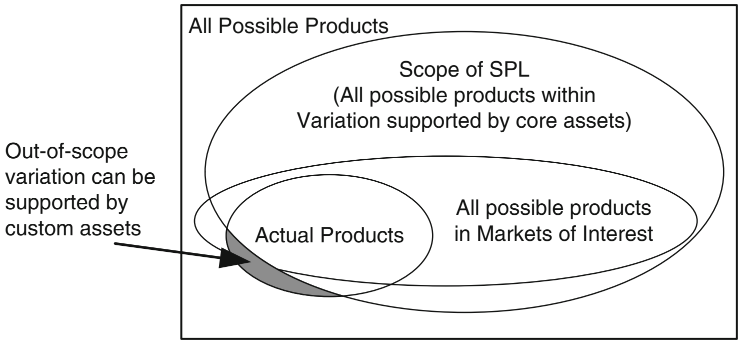
All Possible Products
Scope of SPL (All possible products within Variation supported by core assets)
Out-of-scope variation can be supported by custom assets
Actual Products
All possible products in Markets of Interest

Fig. 15.3 The scope of an SPL

once. Moreover, although the defect might be initially found in the use of only one product, every product in the SPL will benefit from the defect fix. These factors allow more rapid improvements to product quality in SPL development.

There are additional second-order benefits to SPL development. For example, SPL development provides organizations with a clear path enabling them to turn customized project work for specific customers into product line features reused throughout the SPL. When organizations have processes in place to managed reused assets, the development of customer-specific project work can initially be managed in a custom asset. If the features prove to have wider significance, the custom asset can be moved into the reused core asset base.

Another related benefit is that the management of core and custom assets provides a clear and simple view of the range of products maintained by the organization. This view enables organizations to more easily:

- Upgrade products to use a new core version
- See what assets are core for the business
- See how products differ from each other
- Consider options for future functionality for the SPL

15.2.2 Product Lines for ICDE

The three planned ICDE products all operate in a similar way and the differences for each of the products are fairly well understood. The Government product will have a user interface that supports policy and governance checklists, the Finance product will support continually updated displays of live market information, and the Intelligence product will integrate views of data from various sources of classified data.

The variation required in the product line can be defined largely in terms of the data collection components. The GUI options and the access to domain specific data sources will have to be supported by variation points in the collection components. This means the *Data Collection* client component will need variation points in order to support access to application domain-specific data sources. This will require custom components to handle the specific details of each of the new government/financial/intelligence data sources. The *Data Store* component should not need to support any variation for the three different products. It should be able to be reused as a simple core asset.

15.3 Product Line Architecture

SPL development is usually described as making use of a Product Line Architecture (PLA). A PLA is a reuse-oriented architecture for the core assets in the SPL. The reuse and variation goals of a PLA are to:

- Systematically support a preplanned scope of variant functionality
- Enable products within the SPL to easily choose options from among that variant functionality

A PLA achieves these goals using a variety of technical mechanisms for reuse and variation that are described in the following sections. Jan Bosch[2] has identified three levels of PLA maturity:

1. Under-specified architecture (ad-hoc variation)
2. Specified architecture
3. Enforced architecture (all required variation supported by planned architectural variation points)

Increasing levels of architectural maturity provide more benefits from systematic variation by making product development faster and cheaper. However, increasingly mature PLAs provide fewer opportunities for ad-hoc variation, which can reduce opportunities for reuse. Nonetheless, increasing levels of reuse can be achieved if there is better systematic variation, that is, better adaptation of the PLA to the scope and application domain of the SPL.

A PLA is not always necessary for successful SPL development. The least mature of Bosch's maturity levels is "under-specified architecture," and experiences have been reported of the adoption of SPL development with an extremely under-specified PLA. Although products in an SPL will always have some sort of architecture, it does not necessarily have to be a PLA, namely one designed to support goals of reuse and variation. Essentially, to reuse software, developers must:

1. Find and understand the software
2. Make the software available for use by incorporating it into their development context
3. Use the software by invoking it

Let's look at each of these steps in turn.

15.3.1 Find and Understand Software

Software engineers use API documentation and reference manuals to support the simple reuse of software libraries. For SPL development, the Product Line Practice guidelines from the SEI (see Further Reading) describe the *Product Parts Pattern* which addresses the discovery and understanding of core asset software for SPL development. This pattern relies on the documentation of procedures to use and instantiate core assets in the construction of products.

[2] J. Bosch, *Maturity and Evolution in Software Product Lines*. In Proceedings of the Second International Software Product Line Conference (San Diego, CA, U.S.A., August 19–22 2002). Springer LNCS Vol. 2379, 2002, pp. 257–271.

15.3.2 Bring Software into the Development Context

After finding the software, a developer has to make it available to be used. There are many ways to bring software into a development context, which can be categorized according to their "binding time." This is the time at which the names of reused software assets are bound to a specific implementation. The main binding times and some example mechanisms are:

- Programming time – by version control of source code
- Build time – by version control of static libraries
- Link time – by operating system or virtual machine support for dynamic libraries
- Run time – by middleware or application-specific mechanisms for configuration or dynamic plug-ins, and by programming language mechanisms for reflection

Earlier binding times (such as programming or build time) make it easier to use ad-hoc variation. Later binding times (such as link or run time) delay commitment to specific variants, and so make it easier to benefit from the options provided by systematic variation. Increasingly mature PLAs for SPL development tend to use later binding time mechanisms. This enables them to maximize the benefits from an SPL scope that is well understood and has a good fit with the company's markets of interest.

15.3.3 Invoke Software

To invoke software, programming languages provide procedure/function/method call mechanisms. For distributed systems, interoperation standards such as CORBA and SOAP provide remote invocation mechanisms that are tied into programming language mechanisms, to allow developers to invoke software systems running on other machines. These invocation mechanisms are the same for SPL development as for traditional software development.

15.3.4 Software Configuration Management for Reuse

For organizations that are adopting SPL development, the most common binding times for reuse are programming time and build time. This makes software configuration management (SCM) a critical supporting process area for SPL development. SCM includes version control and change control for software assets.

SCM for SPL development is more complicated than in normal product development partly because configuration identification (CI) is more complicated. CI is the SCM activity of specifying the names, attributes, and relationships between configurations (a versioned collection of versioned objects). In normal product

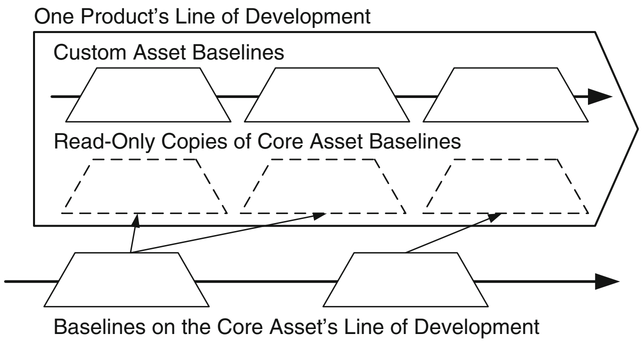
One Product's Line of Development
Custom Asset Baselines
Read-Only Copies of Core Asset Baselines
Baselines on the Core Asset's Line of Development

Fig. 15.4 A SCM branching pattern for SPL development

development, a product's configuration usually has a simple structure (e.g., a single versioned binary or versioned file system directory hierarchy). However in SPL development, each core asset, custom asset, and product is a configuration that must be identified and the relationships between these configurations must be specified and managed. Basically, SCM gets much more architectural for SPL development.

One approach to SCM for SPL development is depicted in Fig. 15.4. In this approach, core assets and products each have their own line of development (LOD). Each product version includes its own custom assets, as well as versions of core assets. The version control system ensures that reused core assets are read-only for a product, and that they are not modified solely within the context of a specific product's LOD. However, a product's LOD can take a later version of a core asset which has been produced on its own LOD.

This view of SPL development provides a quantitative basis for seeing why SPL development can prove so effective. The LOD for each product contains source code for customer-specific assets and also (read-only) source code for core assets. So each LOD contains essentially the same source code as it would were product line approaches not being used. However the total volume of branched code has been reduced, because the size of core assets is not multiplied across every product. Core assets are not branched for each product, and so low level design, coding and unit test costs within core assets can be shared across many products.

In the ICDE example there are three products, and let's assume that the core components have 140,000 LOC (Lines of Code) and each product's custom part have 10,000 LOC. In normal product development, each product would be maintained on a separate LOD, giving a total of:

$$(140,000 + 10,000) \times 3 = 450,000 \text{ branched LOC.}$$

In SPL development, the core is on its own LOD, and each product has a LOD only for changing their custom assets, giving a total of:

$$140,000 + (10,000 \times 3) = 170,000 \text{ branched LOC.}$$

That's only 38% of the original total. The improvement gets better when developing more products, or when the size of the custom assets compared to core assets is proportionately smaller.

15.4 Variation Mechanisms

In an SPL, core assets support variable functionality by providing variation points. A PLA typically uses specific architectural variation mechanisms to implement variable functionality. However, an SPL can also use nonarchitectural variation mechanisms to vary software functionality.

In addition to architectural-level variation mechanisms, there are design-level and source-level variation mechanisms. These different types of variation are not incompatible. For example, it is possible to use file-level variation at the same time as architectural variation. This section describes some of the variation mechanisms at these different levels of abstraction. This classification is similar to the taxonomy of variability realization techniques in terms of software entities that has been proposed by Svahnberg et al.[3]

15.4.1 Architecture-Level Variation Points

Architectural variation mechanisms are high-level design strategies intended to let systems support a range of functionality. These strategies are only very loosely related to the facilities of any specific programming language. Examples of these include frameworks and plug-in architectures. Even the formal recognition of a space of configuration options or parameters for selecting between variant functionality can be considered to be an architectural variation mechanism.

15.4.2 Design-Level Variation

The boundary between architecture and design is not always a clear one. Here we will say that design-level mechanisms are those supported directly by programming language facilities and that architecture-level mechanisms must be created by programming. Programming language mechanisms can be used to represent variation. These mechanisms include component interfaces that can allow various functionally different implementations, and inheritance and overriding that similarly allow objects to have variant functionality that satisfies base classes.

[3]M. Svahnberg, J. van Gurp, J. Bosch, *A Taxonomy of Variability Realization Techniques*, Technical paper, Blekinge Institute of Technology, Sweden, 2002.

15.4.3 File-Level Variation

Development environments and programming languages provide ways to implement variation at the level of source code files. Some programming languages provide conditional compilation or macro mechanisms that can implement functional variation. In any event, build scripts can perform logical or physical file variation that can be used to represent functional variation.

15.4.4 Variation by Software Configuration Management

The main role of SCM for product line development is to support asset reuse by identifying and managing the versions of (and changes to) products and their constituent component assets. New product versions do not have to use the most recent version of a core asset. SCM systems can allow a product to use whatever core asset version that meets the needs of the product's stakeholders. The version history and version branching space within an SCM tool can be used to represent variation.

In a version control tool, a branched LOD of a core asset can be created to contain variant functionality. Branching reused core assets in order to introduce ongoing variation is a sort of technical decay that reduces the benefits of SPL development. In the extreme case where every product has its own branch of core assets, an organization will have voided SPL development completely and will be back doing ordinary product development. Nonetheless, in some circumstances a temporary branch is the most pragmatic way to introduce variation into a component in the face of a looming delivery deadline.

15.4.5 Product Line Architecture for ICDE

Early on in the development of the ICDE product the development team had put considerable effort into the product architecture. This means that they're in the fortunate position of already having many architectural variation mechanisms in place, making the adoption of product line development easier. For example, the *Data Source* adapter mechanism provides all the required variability for the three new products. These existing variation mechanisms form the heart of the product line architecture for the ICDE product line.

The team needs to define some new variation mechanisms too. To support the real-time display of market information for the Financial product, the existing GUI components need new functionality. The GUI is currently too rigid, so the team plans to extend the GUI framework to let them add new types of "plug-in" panels connected to data sources. When this framework is extended, it'll be much easier to implement the real-time display panel, connect it to the market data source, and include it in the GUI for the Financial product build.

However, although the ICDE team thought the *Data Store* would be the same for all three products, it turns out that separating the classified data for the Security product is a nontrivial problem, with requirements quite different from the other two products. The team has to come up with some special-purpose *Data Store* code just for that product. The easiest way to make these special changes is in a separate copy of the code, so in their version control tool they create a branch of the *Data Store* component just for the Security product. Having to maintain two different implementations of the *Data Store* might hurt a little, but it's the best the team can do under a tight deadline. Once the product ships they'll have time to design a better architectural variation mechanism for the next release, and move all the products onto that new *Data Store* component.

15.5 Adopting Software Product Line Development

Like many radical business changes, the adoption of SPL development in an organization is often driven in response to a crisis (what Schmid and Verlage[4] diplomatically called a "reengineering-driven" situation). This may be an urgent demand to quickly develop many new products, or to reduce development costs, or to scale new feature development in the face of a growing maintenance burden. This section points out some paths and processes relevant to the adoption of SPL development.

There are two different starting points in the adoption of SPL development:

1. *Green Fields*: where no products initially exist
2. *Ploughed Fields*: where a collection of related legacy products have already been developed without reuse in mind

Each situation has special considerations, as described below.

For Green Fields adoption of product lines, the SEI's *What to Build* pattern is particularly relevant. This pattern describes how a number of interacting practice areas can result in the generation of an SPL Scope (to know what SPL will be built) and a business case (to know why building the SPL is a good investment for the organization). The SEI's *Scoping* and *Building a Business Case* practice areas that are directly responsible for these outputs are supported by the *Understanding Relevant Domains*, *Market Analysis*, and *Technology Forecasting* practice areas.

An organization has to decide on their markets of interest, their medium-to-long term SPL scope, and their short-to-medium term product production plans. The organization must plan and evaluate the various investment options of having the PLA of the core asset base support a large-enough SPL scope. This makes it

[4]K. Schmid, M. Verlage, *The Economic Impact of Product Line Adoption and Evolution*. In IEEE Software, July/August 2002, pp. 50–57.

possible to trade off the potential for return from the products that can be generated within that scope for the markets of interest to the organization.

Investing in a PLA at the beginning of an SPL will provide a better long-term return assuming that the products in the SPL are successful in the market. However, the cost and technical difficulty of creating such a PLA *ex nihlio* can pose a barrier to the adoption of SPL development, especially if the organization is not already expert within the application domain being targeted by the SPL.

In contrast, when a set of products exists and is being transitioned to an SPL, an organization will, as for Green Fields adoption, need to decide on the SPL scope and markets of interest for the SPL. However, organizations in this position will generally already have a good understanding about these. The scope of the SPL will largely be driven by the functionality of existing products and future product plans. The other significant considerations for Ploughed Fields adoption are potential barriers related to change control, and defining the core assets and PLA.

Change control issues can pose a barrier to the adoption of SPL development for an organization's legacy products. The stakeholders of existing products will already have established expectations about how their product releases change. As discussed in the SCM section, every product in the SPL has stakeholders that influence changes made to core assets, and these core asset changes in the SPL will ultimately affect every product in the SPL, including other stakeholders. This change in the nature of product releases must be understood and accepted by the products' stakeholders.

When initially defining an SPL for an existing set of independent products, the organization must decide what is core for every product, and what is custom or specific to any individual product. Instead of throwing away the existing assets for the organization's products and starting from a blank slate, it is possible to use an extractive approach to mine core assets from existing products. The SEI describes a product line practice area *Mining Existing Assets* addressing this activity. In many ways, the extraction of core assets is like a giant refactoring exercise, as depicted in Fig. 15.5. Starting from an initial collection of products, the goal of

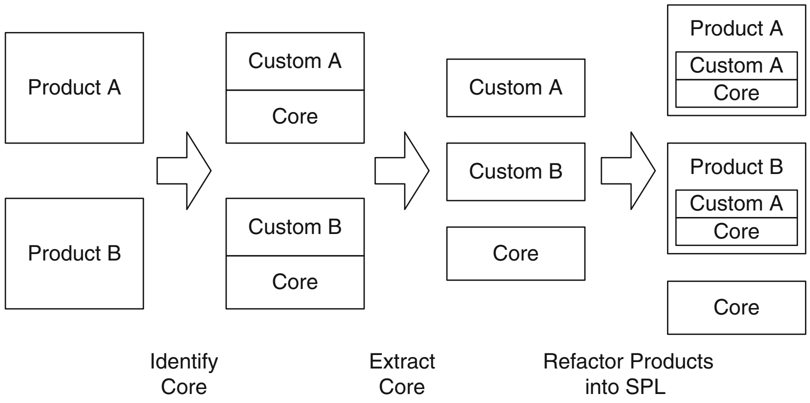

Fig. 15.5 Mining core assets from a collection of existing products

the exercise is to finish with identical products, except now all built using a common core asset.

When defining the core assets, the organization can also define a PLA to cater for variation that is identified among the products. Svahnberg et al. have presented a set of minimally necessary steps to introduce variability into a SPL. These are:

- Identification of variability
- Constraining variability
- Implementing variability
- Managing the variability

In order to reduce change control conflicts, it may be easier to introduce SPL development early in the cycle leading to the release of a major new version of a product. Product stakeholders are prepared for major changes when receiving a major new version. Although moving to SPL development need not in principle result in any functional difference to a product, there will at least be change control policy modifications, which customers may find easier to accept in the context of a major new product version.

An organization adopting product lines can also reduce business and technical risks by incrementally rolling out the SPL within the organization. Adoption can be incremental either by progressively increasing the size of the core assets, by progressively adding more products to use the core assets, or a combination of both.

15.5.1 Product Line Adoption Practice Areas

The adoption of SPL development has impact outside the technical development context. Regardless of the starting point for product line adoption (Green or Ploughed Fields) and regardless of the specific product and technical process changes that are to be made, many organizational management issues must be dealt with to successfully transition to SPL development. The SEI product line practice guidelines describe the *Cold Start Pattern* that groups together practice areas that can help an organization effectively prepare for the launch of its first SPL. The structure of the pattern is shown in Fig. 15.6.

Although the details of these practice areas are beyond the scope of this chapter, the pattern as a whole highlights the fact that SPL development must have broad business support from within the adopting organization and from its customers.

15.5.2 Product Line Adoption for ICDE

The ICDE team was driven to SPL development by the daunting prospect of developing three new products at once. They are creating three new products for three specific markets, but are using their existing product as a starting point.

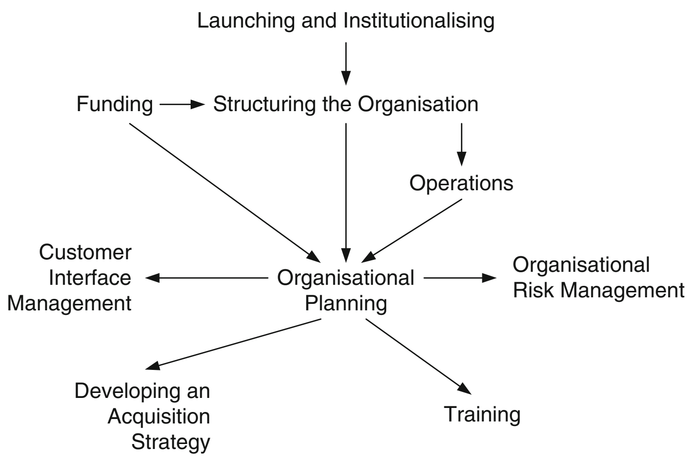
Launching and Institutionalising
Funding
Structuring the Organisation
Operations
Customer Interface Management
Organisational Planning
Organisational Risk Management
Developing an Acquisition Strategy
Training

Fig. 15.6 The structure of product line practice areas in SEI's *Cold Start* pattern (after Clements and Northrup 2002, p383)

Their adoption of SPL development is thus a Ploughed Field scenario. They have to mine reusable components from their existing code base.

Luckily their existing customers aren't going to be too concerned initially about the move to a PLA, because the move is part of the development of a major new version of the product. The customers will be happy to upgrade because of the new features they'll also be getting.

15.6 Ongoing Software Product Line Development

SPL development must be effective not just for the initial development of new products, but also for their ongoing maintenance and enhancement. Although SPL development can have many benefits, it is more complicated than normal product development. Enhanced processes are necessary to make ongoing SPL development effective. This section gives an overview of a few of these SPL development processes. We pay particular attention to "change control" and "architectural evolution" for SPL development, but also summarize other SEI Product Line Practice areas for ongoing SPL development.

15.6.1 Change Control

Software change control is related to software configuration management, and is concerned with planning, coordinating, tracking, and managing the impact of change to software artifacts (e.g., source code). Change control is harder when you do software reuse, and this affects SPL development.

In any kind of product development, every product has a collection of stakeholders that is concerned with how their product changes to accommodate their needs for new functionality. In addition, stakeholders are concerned about nonfunctional characteristics (such as release schedule, product reliability) related to the release of their products. Risk-averse stakeholders (such as those using safety-critical software or those in the banking industry) are often motivated to ensure that their products do not change at all! Such stakeholders sometimes prefer to be confident in their understanding of the product (bugs and all) rather than use new, perhaps better versions.

Change control is harder when you do software reuse, including software reuse for SPL development. For ordinary product development, each product is developed separately, and so each product's stakeholders are kept separate too. However, in SPL development each product depends on reused core assets, and so these products' stakeholders also vicariously depend on these reused core assets. If one product's customer has a change request that involves a change to a core asset, then implementing that will force that change on every other customer who uses the new version of that core asset. The many, often conflicting, needs of the products' stakeholders will need to be simultaneously satisfied by the reused core assets.

15.6.2 Architectural Evolution for SPL Development

In SPL development there is constant evolution of both individual product custom assets and the reused core assets. The PLA is the architectural basis for the variation supported by core assets. A change to a core assets' interface is a change to the PLA, and can force changes in all products that use the new version of these core assets. How then should the new or enhanced core features be added to a product line? That is, how should changes be made to the PLA?

There are three ways to time the introduction of variation points into core assets:

- *Proactive*: Plan ahead for future features, and implement them in core assets before any product needs them.
- *Reactive*: Wait until a new feature is actually required by a product, and then implement it in core assets at that time.
- *Retroactive*: Wait until a new feature is actually required by a product, and then implement it in a custom asset at that time. When enough products implement the feature in their custom assets, add it to the core assets. New products can use the new core assets' feature, and the older products can drop their custom asset implementation in favor of the core assets' implementation.

It is possible to use a mix of these approaches, for different enhancements. For example, enhancements on a long-term Road Map could be added in a proactive way, by planning architectural changes to support the future increased scope of the SPL. Limited but generally useful enhancements to core assets could be added in a reactive way, by modifying the PLA as required by those enhancements.

Table 15.1 Comparing strategies for architecture evolution

	Proactive	Reactive	Retroactive
No long-term investment	No	Yes	Yes
Reduces risk of core asset change conflict	Yes	No	Yes
Reduces lead time to add feature to first product	Yes	No	No
Reduces risk of core feature not required in a number of products	No (0 products)	No (1 product)	Yes

Enhancements needed by one product that are more speculative or are less well defined could be added retroactively.

Each of these strategies has different costs, benefits, and risks. The choice of strategy for a particular feature will be driven by consideration of these tradeoffs in the organization's business context. Table 15.1 summarizes some of the differences between the three approaches:

15.6.3 Product Line Development Practice Areas

The SEI product line practice guidelines provide the *Factory* pattern that links together other patterns and their constituent practice areas relevant to the ongoing development and maintenance of a SPL. The *In Motion* pattern groups together organizational management practice areas. Other relevant SEI patterns are the *Monitor*, *Process*, and *Curriculum* patterns that describe ongoing aspects of SPL development.

For technical practice areas, the SEI's *Each Asset* pattern describes practice areas that are relevant to the development of core assets. The *Product Parts* pattern ties together the core assets with the product development. The *Product Builder* pattern describes practice areas relevant to the development of any specific product. The *Assembly Line* pattern describes how products are output from the SPL.

15.6.4 Product Lines with ICDE

Doing SPL development wasn't just an architectural issue for the ICDE team. Each of the products had a customer steering group that was involved in defining requirements for the new products, and defined enhancement requests that they wanted to track through to the delivery of the products. But the ICDE team didn't want the Financial product customer steering group to see all the details of the Security product steering group, and vice-versa. The problem was that some enhancement requests were the same (or similar), and the team didn't want to get confused about duplicate requests when they started coding.

So, the ICDE team set up different customer-facing request systems for each of the products. These linked to an internal change request system which could track changes to each of the main reused subsystems and also the product-specific custom components.

Eventually the first product was released. Instead of releasing all three products at once, the team shipped the simplest product first, namely the Government product. The Government customers quickly raised a few postrelease defect reports, but the ICDE development team was able to respond quickly. The good news was that one of the defects that was fixed was in the core *Data Collection* component, so when the other two products were released later, their customers wouldn't see that problem. The ICDE team was beginning to see some quality benefits from SPL development.

The bad news came after the other products were released. The Security and Financial customers were happy to have the new version, though the Financial customers did raise a defect report on the *Data Analysis* component. It would have been easy to fix in the core component, but by that time the Government customers had gone into production. They hadn't seen that problem in the *Data Analysis* area, and in fact the bug was related to the framework extensions required to support the Financial product real-time display panel.

However, if the *Data Analysis* component changed in any way at all, the Government customers would have to follow their policy and rerun all of the related acceptance tests, which would cost them time and money. So they really didn't want to see any changes, and put pressure on the ICDE sales team to try to stop the change.

The ICDE development team really wanted to change the core version, but how could they satisfy everyone? They thought about faking the core changes in custom assets just for the Financial product, but in the end they decided to keep the Government product on the old version of the *Data Analysis* component, and implemented the fix in the core. The ICDE development team also created a Core CCB involving representative members from each of the three customer steering groups. This meant that in future the negotiations could be managed inside the Core CCB, instead of via the ICDE sales team.

A bright spot on the horizon was that the Security customers were starting to talk about their need to see real-time visualization of news reports. The ICDE development team could implement that just by reusing the real-time display panel developed for the Financial product. The company had already accounted for the costs of developing that feature, so being able to sell it again to other customers would mean all the new revenue would go straight to the bottom line.

15.7 Conclusions

Product line development has already given many organizations orders of magnitude improvements to productivity and time to market, and significant improvements in product quality. If we think about SPL development simply from a SCM perspective, we can see that (proportionately large) core assets are not branched for each product, and so the total number of branched lines of code is vastly reduced for the whole SPL.

What does the future hold for SPL development? Because of its massive potential, SPL development is likely to become even more widely known, better understood, and increasingly used. However, SPL development will also have impacts on software architecture practices, as architectural mechanisms for reuse in the large become better and more widely understood.

Improved architectural practices combined with a deeper understanding of specific application domains can also support increasingly declarative variation mechanisms. This could transform software reuse to be more like the mythical vision of software construction using software building blocks. Simple reuse relies heavily on procedural variation, writing ad-hoc code to achieve the particular functionality that is required. Increasing architectural sophistication and domain knowledge can support configurable variation, realized by systematic variation supported by core assets interfaces.

Choosing a variant for such a system requires choosing values from a list of configuration options. When an application domain is very well understood, then a domain-specific language becomes a viable way of declaratively specifying product variation. Sentences in this language can specify system variants, and can be dynamically interpreted by the core assets.

Other architectural and design approaches such as aspect-oriented programming and model-driven development also have promise as variation or mass-customization mechanisms that may be able to support SPL development.

As the time of system variation extends out of the development context, so does the need to extend the control and management of variation. For systems that can vary at installation time, load time, or run time, the need to control and manage system variation does not end when the system is released from development. Software configuration management supports control and management of variation during development. However, for installation, load or run time, existing package management and application management frameworks have very weak facilities for version and variation control. In future, the boundaries between configuration management, package management, and application management will become blurred. A unified framework is therefore required to control and manage variation across the entire product lifecycle.

15.8 Further Reading

The Software Engineering Institute has been a leader in defining and reporting the use of software product lines. An excellent source of information is the following book by two of the pioneers of the field:

P. Clements, L. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley, 2001.

The SEI's web site also contains much valuable information and links to other product line related sources:

http://www.sei.cmu.edu/productlines/

Other excellent references are:

Klaus Pohl, Günter Böckle, Frank J. van der Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer-Verlag 2010

Frank J. van der Linden, Klaus Schmid, Eelco Rommes, Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering, Springer-Verlag 2007.

Software configuration management is a key part of software product lines. A good book on this topic is:

S.P. Berczuk, B. Appleton. Software Configuration Management Patterns: Effective Teamwork, Practical Integration. Addison-Wesley, 2002.

A case study describing how to exploit file-based variation to create a software product line is:

M. Staples, D. Hill. *Experiences Adopting Software Product Line Development without a Product Line Architecture*. Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC 2004), Busan, S. Korea, 30 Nov – 3 Dec 2004, IEEE, pp. 176–183.

A slightly different perspective on product lines is the Software Factories work by Jack Greenfield et al. This book is definitely worth a read.

J. Greenfield, K. Short, S. Cook, S. Kent, J. Crupi, Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, Wiley 2004.