Variability in Software Product Lines

Felix Bachmann
Paul C. Clements

September 2005

TECHNICAL REPORT
CMU/SEI-2005-TR-012
ESC-TR-2005-012

Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

Variability in Software Product Lines

CMU/SEI-2005-TR-012
ESC-TR-2005-012

Felix Bachmann
Paul C. Clements

September 2005

Product Line Practice Initiative

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

List of Figures

List of Tables

Acknowledgments

The authors thank Linda Northrop, Charles Krueger, and Rob van Ommering, as well as John McGregor and Gary Chastek, for thoughtful, insightful reviews that led to substantial improvements to this report.

Abstract

Product line engineering is a widely used approach for the efficient development of whole portfolios of software products. The basis of the approach is that products are built from a *core asset base*, a collection of artifacts that have been designed specifically for use across the portfolio. To account for differences among the software products, some adaptations of the core assets are usually required. These adaptations should be planned before development and made easy for the product developers to use without jeopardizing existing properties of the core assets.

In a product line with a large number of products and core assets, as well as requirements to make fine-grained adjustments, managing variability can become problematic very quickly. Mismanagement may result in adding unnecessary variability, implementing variation mechanisms more than once, selecting incompatible or awkward variation mechanisms, and missing required variations. As the product line grows and evolves, the need for variability increases, and managing the variability grows increasingly difficult.

This report describes the concepts needed when creating core assets with included variability. These concepts provide guidelines to core asset creators on how to model the variability explicitly, so it is handled consistently throughout the product line and managing the variability becomes feasible.

1 Introduction

Product line engineering has become an important and widely used approach for the efficient development of whole portfolios of software products [McGregor 02]. The fundamental idea of the approach is to undertake the development of a set of products as a single, coherent development activity. Products are built from a *core asset base*, a collection of artifacts that have been specifically designed for use across the portfolio. This approach has been shown to enable order-of-magnitude improvements in quality, time to market, cost, and productivity, compared to one-at-a-time software system development [Clements 02b].

Core assets include, but are not limited to, the architecture and its documentation, specifications, software components, tools such as component or application generators, performance models, schedules, budgets, test plans, test cases, work plans, and process descriptions [Clements 02b]. Although it may be possible to create core assets that can be used across the products without any adaptations, in many cases some adaptations are required to make core assets usable in the broader context of a product line. Variation mechanisms used in core assets help to control the required adaptations and to support the product developers in their task.

The following example will help us illustrate the concept of variation mechanisms. In a product line of software to support bank loan offices, suppose we have a software component in the core asset base that calculates what a customer owes in the current month. For 18 of the 21 products, this component is completely adequate. However, three of the products will be used in banks in the state of Delaware, which has certain laws that affect what a customer can owe. The difference affects 25 lines of source code in our 8,000-line component. Clearly, we would rather not copy the component, change the 25 lines, and deploy a new component, because then there would be two (almost identical) 8,000-line components to maintain as the product line evolves; this so-called “clone and own” strategy quickly gets out of hand. We would much rather find a way to take advantage of the fact that these nearly identical components vary only in a small, well-defined way. To take advantage of their similarities, we introduce a variation mechanism into the component. The effect is as though the core asset developer installed a switch for the product developer to use. When the switch is flipped one way, we get the version of the component that will go into the Delaware products. When it is flipped the other way, we get the version of the component that will go into all the other products. This variation mechanism will let us maintain a single component that can adapt to the range of variations in the applications that it has to support. We may need versions for other states in the future, so our switch—that is, our variation mechanism—should have the ability to accommodate those possibilities as well.

Developing a core asset base requires the following major activities:

1. determining what about the core asset can remain the same for all of the products in which it will be used and what has to vary from product to product
2. choosing a variation mechanism that supports the variation required, while allowing the commonality to be provided without any change from product to product
3. providing instructions (in the form of a *production plan* [Chastek 02]) that explain to product developers how they must use the variation mechanisms included in the core asset to create a product

Although we briefly discuss production plans throughout this report, we are mainly concerned with the first two activities: (1) how core asset developers determine what has to vary from product to product and (2) how to choose variation mechanisms to employ.

The remainder of this report focuses on how to make educated decisions on what type of variation mechanisms to include in core assets. The next section establishes some basic concepts that will be useful in discussing the selection of variation mechanisms. Section 3 asserts that the goal for choosing a variation mechanism is economic in nature and will guide the choice of the variation mechanism. Section 4 describes three broad factors to consider when choosing a variation mechanism: available variation mechanisms, product information, and production strategy. Section 5 introduces variability scenarios—a way to conveniently express the requirements for variability imposed by the product line's product set and production strategy. Section 6 introduces a small selection of variation mechanisms and describes how they help to achieve variability goals. Section 7 contains an outline of a variability management process that shows how the variability in all the core assets of a product line can be managed in a practical way. Section 8 describes related work, and Section 9 summarizes and proposes future work.

2 Variability Concepts

Variability is the ability of a system, an asset, or a development environment to support the production of a set of artifacts that differ from each other in a preplanned fashion. For the purposes of this report, variability means the ability of a core asset to adapt to usages in the different product contexts that are within the product line scope. Implicit in this definition is the fact that variations in a product line context are anticipated. The developers of core assets have thought about the consequences of the variations and, presumably, constrained them in a way that the resulting core assets support the requirements and take into consideration the time and budget available to create the assets.

When preparing core assets to support product-by-product variation, the concepts described in Sections 2.1–2.6 are useful.

2.1 Core Assets and Product Assets

A core asset is an artifact or resource that is built to be used in the production of more than one product in a software product line. As such, a core asset is used in the production of a product asset. A product asset is an artifact that is part of a product[1] in a software product line. Core assets might be software components, the architecture and its documentation, budgets, schedules, plans, user manuals, test plans, tools, process definitions, analysis models, configuration management plans, interface specifications, and a myriad of other things. Anything that is used in the creation of a product is considered a legitimate core asset—including all artifacts that are associated with producing the product [Clements 05a], which go well beyond just the product's deliverable software.

The mapping between core assets and product assets is sometimes, but not always, one to one. Here are several illustrations of that point, using different variation mechanisms:

- copy unchanged: Some core assets are used in a product without change. In this case, the core asset either handles no variation and its scope of variation is nil, or it handles variation by making runtime choices (i.e., via if-then-else statements). In this case, a core asset is identical to a product asset.

[1] Some schools of thought require that the asset be used in two or more products before it is considered a legitimate core asset. Others assert that an asset potentially (if not actually) used in more than one product should be treated as a core asset. It is an interesting and useful debate, but it is irrelevant to the topic of this report.

- configurator: A configurator tool can build a source file for a software component by concatenating together blocks of source code that are stored in the core asset base as separate files. Some blocks represent common parts; others are segments that implement product-specific behavior. In this example, each block is a core asset—maintained and stored separately—as is the configurator tool itself. When a set of blocks for a particular product are concatenated together, the resulting source file is a product asset. (Configurators also work well for non-software assets such as documents.)
- generator: A component generator takes a specification of the desired component as input and produces the component as output. In addition to the generator, a users' manual and a tool for analyzing the generated code are core assets that are used in concert to create a software component (a single asset) for a particular product. The product asset is the component produced by the generator.
- inheritance: A core asset base might include several files of class definitions, from which a product builder can select for inclusion in a component, and then extend via inheritance. The product asset is the component with the extensions.
- macro-expansion: A software component in the core asset base contains #ifdef statements around code blocks that are turned "on" or "off" by a compiler preprocessor. The output of the preprocessor that instantiates a core asset is the product asset, but it only exists during the execution of the compilation step and cannot be seen or manipulated by the product developer.

2.2 Selection and Modification vs. Creation

Product developers use core assets to create product assets in one of two ways: (1) selection and modification, or (2) creation.

With selection and modification, a core asset is selected from the core asset base and modified or configured in some preplanned fashion. The resulting product asset is still recognizable as a variant of the original core asset. Variation mechanisms that employ selection and modification include

- verbatim reuse or "copy without change" (The modification is null.)
- parameterization; for example, to allow variation in
  - values of Booleans/integers
  - more complicated data structures
  - types
  - segments of code
  - function
  - objects/classes
  - component instances/components
- configurator (assembling whole product assets by putting together pieces that are core assets)

- component substitution (selecting from existing variants and inserting into core assets)
  - aspects (selecting and inserting either at precompile or compile time)
  - code components (selecting and inserting at compile time)
  - plug-ins (selecting and inserting at runtime)
  - selection of services in a service-oriented architecture
- templates (filling in product-specific parts in a generic body)[2]
- inheritance (the definitions of classes that are used in the product and inherited from generic classes defined for the product line)

With creation, a core asset is used to create a product asset that does not resemble the core asset(s) that spawned it. A generator is the exemplar of this category: The generated result resembles neither the generator nor the input provided to it.

Jan Bosch distinguishes between "open" variability (with room for new variants) and "closed" variability (in which the number of variants is fixed) [Bosch 00]. That criterion can be applied easily to the preceding list to determine which mechanisms are open versus closed (although parameterization could fall into both categories).

2.3 Isolation

Isolation is a concept that applies to core assets that are selected and modified. The goal is to limit where modifications are made to a few well-defined places—ideally, a single place. These places are called *variable parts*.[3] Everything else is a *common part*. Common parts will not change as the core asset is used from product to product. The variable part might be empty initially until the requirements for a specific product are known. This separation of variable and common parts tends to minimize the amount of information (code, if the core asset is software) that has to change. It also helps in testing because tests that apply only to the common parts may only need to be performed once, rather than once per product.

2.4 Variants and Product Assets

A variable part is the place in an asset that is allowed to vary. When a core asset is created, everything that is required for the common part is produced and filled in. The variable part might be empty (for instance, with inheritance), or it might be expressed as alternatives or uninstantiated transformational specifications (as with parameterization, runtime variation, or configurators). Eventually, however, the variable part is also produced, its variation is bound

---

[2] A template can also be viewed as a kind of parameterization, where the product-specific parts are bound to the parameter value.

[3] Some authors refer to these *variable parts* as *variation points*. We have chosen not to use that term in this report for two reasons. First, a variable part not only describes a location (a point) in the core asset that needs adaptations, but it is also an organizing container for all the artifacts (such as variation mechanisms, process descriptions, and variants) that are used to make product-specific adaptations. Second, *variation point* is often used to describe variations in terms that refer to an asset's externally visible properties or functions rather than places in the asset's internal structure.

(the “switch” is “flipped,” to return to the example from Section 1), and the resulting product asset is integrated into the product. The realization of a variable part, meaning the result of exercising the variation mechanism(s), is called a *variant*. A core asset can lead to many variants and many potential variants. The actual (as opposed to potential) variants were presumably created to be product assets.

2.5 Attached Processes

Every core asset should have a statement defined for it that describes how the core asset is to be used in a product (and that implicitly informs a product developer whether the core asset is to be used in a particular product). This statement can be thought of (and is sometimes written) as a condition/process pair, such as the following:

Condition: *If the feature X should be included in the product being produced…*
Process: *…then put the line “xxxyyyzzz” into the configuration file…*

where the process part is to be executed (by a product build tool if available or a product developer if not) if the condition part is true. The process combined with the condition is described as the *attached processes* of core assets in a product line [Clements 02b]. The process describes how the core asset’s variation mechanism(s) should be used to make the required variants for use in products. For example, if inheritance was chosen as a variation mechanism, the core asset will include some class definitions that should be used to implement product-specific classes. The attached process would describe how to make those adaptations without violating design and implementation principles of the core asset.

Real attached processes most likely are more complicated than the example given above. They are also likely to be machine readable and executable, as are the constraints to avoid violating design principles such as avoiding building products with incompatible features. These constraints are executable expressions that are automatically checked during the mapping from feature decisions (“condition”) to instantiations (“process”).

2.6 Dependencies

The *condition* part of an attached process is a description of the dependencies on the values that depict a specific product. For example, a condition can specify that a certain feature has to be present or an input provided by a product developer has to be a certain value. Conditions may have the following dependencies:

- A condition’s value may depend on another variation in the same core asset. For example, the attached process for an architecture says that Component B must be included if Component A is included, where including Component A is predicated on its own condition.
- A condition’s value depends on another variation in a different core asset. For example, the attached process for creating a product-specific user guide may state that a specific

section must be included if a certain feature (as defined in a requirements document) is part of the product.

- A condition's value depends on input from a person. For example, the attached process for a test plan calls for including a test case in the test suite if a tester selects that suite.

Every variant has a value assigned. This value is set when a specific product is being built and reflects the choice made for this variable part. It corresponds to the *condition* part of an attached process. The value could contain information about the existing variant chosen or whether a new variant has to be produced. This value can subsequently be used in conditions of attached processes to influence choices that have to be made elsewhere. For example, if one attached process results in the condition of choosing to include Component A, another attached process could express the following in its *condition* part before giving the resulting condition: "If Component A has been chosen…."

Dependencies (such as "requires," "excludes," or "choose between") are most usefully specified among features, as opposed to among parts of the building blocks of products. Doing so prevents worrying about combinations of product parts in which nobody will ever take an interest.

2.7 Conditional Core Assets

So far, we have discussed the dependencies between variable parts and have ignored the fact that at least some of the core assets themselves are conditional. It is very likely that only a subset of the available core assets is used in building a specific product. This means that we also need to specify under which condition a core asset is included in a product. Then, if that condition is true, the variable parts of that asset might have to be adapted, depending on their condition.[4]

Typically the condition of a core asset depends on some input that describes the required product configuration (e.g., product for WinCE platform, which requires including all the WinCE specific libraries). A condition might also depend on the fact that another asset is included (e.g., if we include the WinCE libraries, we also have to include the WinCE installation files).

Dependencies among variations create a network of relations among the core assets used in a product line, which may be challenging for an organization to manage. Section 7 will return to this topic.

[4] Charles Krueger refers to this step as the "composition and configuration" of core assets [Krueger 05].

2.8 Essential Variability vs. Local Variability

Essential variability describes the ways in which products are required to differ and is determined by a scoping exercise or requirements analysis for the product line. An example of an essential variability is to have an email client included in a high-end desktop product but not in a low-end one. All groups of people who create core assets, such as the component developers or the technical writers, must understand those essential variabilities and prepare their core assets accordingly. Each essential variability should be specifically justified as having compelling reasons (usually related eventually to profitability) for its inclusion. Those essential variabilities must be kept to a minimum and used consistently across all the core assets developed by the different groups.

Local variability is everything else. Core assets developers, such as the people developing test cases, will often introduce additional variabilities to their core assets to achieve the required production qualities. For example, if the developers of test cases introduce an additional variability to use different test harnesses, that is a purely local decision, and no one else, such as a technical writer, should even be aware of this variability. In general, local variabilities should not create any dependencies to other core assets outside the local scope.

Figure 1 summarizes the discussion of this section. Appendix A provides a detailed explanation of this figure.

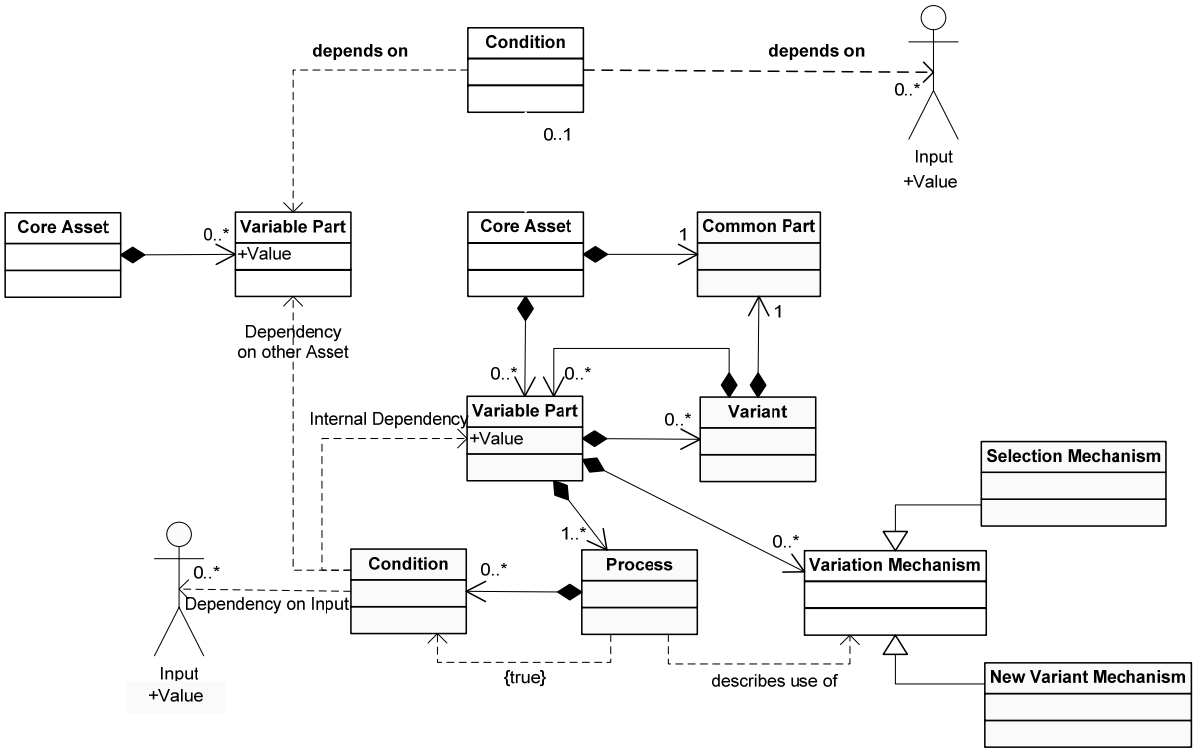


Figure 1: Variability in Product Lines

3 The Goal of Variability

Before we can write meaningfully about choosing the right variation mechanism(s) for a core asset, we must clarify what the goal of variability is—that is, what makes a good choice and what makes a poor choice. Clearly it is desirable to enable the rapid, cost-effective production of products. However, having that as the only goal is not always prudent. For example, Clements and Northrop describe how one product line organization learned a difficult lesson about the tradeoff between making products easy to build and making core assets themselves easy to build:

> The conventional wisdom in software product line production is that the core assets should be tailorable using built-in… mechanisms which provide variability for a wide range of application instances. Component generators exemplify this philosophy; the use of configuration parameters is a widely adopted variant. The appeal is obvious. Product building becomes a matter of "generate and integrate," with a minimum of source code writing.
>
> Salion knew from the beginning that its products would have to be variable over several dimensions. For example, every customer was going to want to use customized forms, so Salion set about writing generic forms software. Given this infrastructure and a statement of a customer's reporting needs, Salion could quickly generate or instantiate a forms component that would satisfy the requirements.
>
> What sounded like a good idea turned out in practice to be a keen disappointment. After eight months of development time, Salion had a generic forms module that didn't work, was much too overblown for what was actually needed, and failed to help the company meet the real requirements of its customers. Recently, the whole thing was thrown out.
>
> Salion experienced the same disappointment and frustration with customized reports. Trying to build a generic component in the absence of actual customer requirements resulted in wasted time and a technical failure.
>
> These and similar lessons have given Salion a completely new outlook on achieving variation. Now its first inclination is to provide variation through customization, not configuration. This means that Salion will modify or rewrite the necessary core assets to produce a version of the system that satisfies a particular customer. As it gains more knowledge about a particular domain, it may eventually be able to design a configurable (or even generative) infrastructure for the

> domain. For instance, of the three major variation points of Salion's products (forms and screens, reporting and export, and bulk load), the company suspects that there's a framework lurking in the first two, which may one day be teased out in a practical fashion. But for now, customization is the watchword [Clements 02a].

Salion's customizable core assets were the right solution for the situation at hand, which was a domain that was too complex to be conquered rapidly via generic or configurable software. This approach does not mean that these assets are developed from scratch for each new product. They carry over common designs, interfaces, architectural interconnection mechanisms, general requirements, test cases, much of the code, and a host of other information that remains the same for each reimplementation of the product.

This example illustrates the difference between the *cost to build* variability into a core asset and the *cost to exercise* that variability. In Salion's case, building the anticipated variability into the software directly turned out to be prohibitively expensive, but exercising that variability would have been quite inexpensive (if they had succeeded). The solution they adopted, which was to customize a standard version of the asset for each product, was more expensive to exercise, but much less expensive to build.

From these observations, we assert the overall purpose of variability in a software product line:

> *The goal of variability in a software product line is to maximize return on investment (ROI) for building and maintaining products over a specified period of time or number of products.*

The cost of producing products includes the cost of building the core assets on which they're based. So the choice of variation mechanisms will affect both the cost of building core assets and the cost of building products from those core assets. For example, an expensive-to-build variation mechanism (such as a program generator based on a domain-specific language) may be justified for a core asset if its cost to exercise is small *and* there will be many products built from that core asset. Or, it may be justified even if there are only a few products built, but the mechanism enables rapid time to market that brings an economic benefit that justifies the investment. We include the notion of a specified period of time because one might make different variability decisions for a short-term horizon than for a long-term horizon.

What about other goals typical for a product line organization, such as higher quality products? These goals can also be seen in economic terms: Higher quality products might cost more to build than lower quality products, but they also might produce higher return because of increased customer satisfaction, enhanced reputation for quality, fewer defects to fix, smaller number of customer complaints to address, and so forth. The assertion that we wish to maximize ROI for building products still holds.

4 Making Variability Choices

Core asset developers need three broad types of information to choose the right variation mechanisms:

1. **available variation mechanisms:** A variation mechanism is used to produce variants of a core asset in a controlled fashion. Although we are not yet to the point of having true catalogs, variation mechanisms are discussed in many publications. (Several of these mechanisms are introduced in Section 6 of this report). A description of a variation mechanism should contain enough information to allow an informed decision about its use, including
   - the kind of core asset(s) for which the mechanism is or is not appropriate. (For example, compile-time selection may be appropriate for core assets that are compiled, but inappropriate for other assets such as the project plan.)
   - the kind of skill and technology necessary to build a core asset using the mechanism
   - the kind of skill and technology necessary to produce a variant (that is, how difficult it will be to use the core asset to build a product)
   - quality attribute considerations for products built with core assets using the mechanism. (For example, might the products have difficulty meeting stringent performance, security, or availability requirements?)
2. **product information:** Product information consists of information known about the products that will be produced from the core assets, especially information about how the products vary from each other. Sometimes this description is a roadmap created by a marketing department, describing the sets of features required for a period of time in the future. Sometimes it is captured in the scope definition for the product line. Quite often it is contained in a detailed feature model for the family of products [Lee 02, Czarnecki 00]. Besides enabling the core asset developer to understand the breadth of the variation among the products, the product information should also contain information about driving quality attributes that might rule out certain variation mechanisms. For example, the need for high performance might rule out the use of interpreters. In addition, beyond looking at the qualities of any product, the complexity (particularly the combinatorics) of the feature model itself may rule out variation mechanisms. In one product line effort, simply keeping numerous (different) versions of a core asset under configuration management was sufficient early on but became unworkable once the product line grew.[5]

[5] This information is from a personal email exchange between Paul Clements and Charles Krueger in 2005.

3. **production strategy:** The production strategy significantly affects how core asset projects are managed and how products are built. It determines, for example, whether an organization builds the product line proactively (starting with a set of core assets and building products from them), builds it reactively (starting with a set of products and generalizing their components to create the core assets), or uses some combination of the two approaches [Clements 05b]. The production strategy also describes when in the life cycle of a product the variabilities included in core assets are exercised and what skill set is required by the people who adapt those assets. The production strategy, therefore, is a source of information to select the appropriate variation mechanisms. For example, in a product line of a supplier for automotive electronic equipment, the organization decided that the inclusion or removal of features had to be done at compile time or link time by developers to minimize resource consumption, such as memory usage, but the fine-tuning of the included features had to be done by service technicians in the field.[6] These constraints were documented in the production strategy and led core asset developers to select conditional compilation for the inclusion of features and configuration parameters for the fine-tuning activity. The production strategy should inform the selection of variation mechanisms based on the following criteria:
   - the budget and schedule constraints for developing a core asset: The budget and time available for building a core asset may lead to a mechanism with a lower cost to build but a higher cost to exercise.
   - the skills and available tool support for the core asset developer: If a developer does not understand object-oriented programming, using abstract classes as a variation mechanism is not appropriate.
   - the skills of the product developer who uses the mechanisms to build a product: If a system administrator has to build a product, nearly all the mechanisms that require programming cannot be chosen.
   - the skills and available tool support for other stakeholders who may be expected (or be given the opportunity) to make variation choices once the product leaves the development shop: These stakeholders include dealers, installers, service technicians, customers, and end users.
   - the resource constraints, such as time and available tools, for product building

If the product variations and the production strategy are not well formulated, the core asset developers may have to choose variation mechanisms without them. In this case, the variation mechanisms may well determine the product variations and production strategy. (That is, the possible product variations and the production strategy will be supported by the variation

[6] Crafting a production strategy is beyond the scope of this report, but the goal of such a strategy can also be seen as the same as that for variability: maximizing ROI for building and maintaining products over a particular period of time. There is a balance between providing the customer with the exact product instantiation that encourages them to buy, being able to get the product instance to them quickly, and ease of instantiation by everyone involved at different variation binding times. Developers of individual core assets are usually not in a position to make these tradeoffs; they should be made by someone whose purview is the overall product line.

mechanisms). However, this is “the tail wagging the dog” because the core asset developers will tend to make choices based on ease of implementation rather than support of the organization’s strategic business goals (which they may not even know). The result will be a large number of different ad hoc variability management techniques adopted over time, with no one individual understanding how they work collectively to instantiate products.

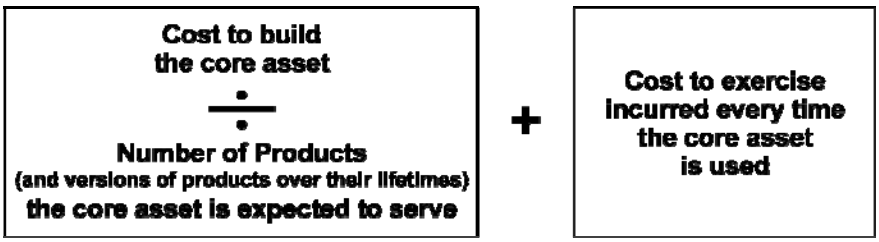
However, the choice of variation mechanisms will still affect the products and production strategy. The variation mechanisms will likely support at least the products chosen and possibly other products as well. The variation mechanisms will likely satisfy the goals of the production strategy, and these goals may even be exceeded. Thus, variation mechanisms are likely to provide additional production capability beyond the minimum required. This capability may represent a new opportunity for the organization, which can then adjust its product set and production strategy accordingly. For example, in the case of an entertainment electronic equipment supplier, core asset developers had to provide Bluetooth connectivity. Instead of just enabling this protocol in a product, the developers created a *generic* protocol that enabled then to integrate entertainment devices easily using different protocols. This new ability to integrate other entertainment devices immediately let the organization expand its product roadmap.

Therefore, the core asset developers should first endeavor to understand the three kinds of information needed. Available variation mechanisms are documented liberally in the software product line literature; several are mentioned in this report. If the product information and production strategy exist, the core asset developers should become familiar with them. If they do not exist, the core asset developers should lobby for their creation.

The selection process then becomes, in theory, a matter of matching the required properties for the mechanisms (as expressed in the product information and production strategy) with the provided properties of the available mechanisms (as expressed in a catalog of such mechanisms). The result will be a list of candidate variation mechanisms. Core asset developers should choose the ones that satisfy the goal (expressed in Section 3) of maximizing the ROI for building and maintaining products over a specified period of time or number of products.

Calculating the cost to build and cost to exercise requires a cost model such as the Structured, Intuitive Model for Product Line Economics (SIMPLE) [Clements 05c]. SIMPLE consists of a set of cost functions that can be used in combination to produce formulas expressing the cost and benefit of taking a particular product line strategy—in this context, the cost and benefit of choosing one kind of variation mechanism over another. SIMPLE formulas can be constructed to express the costs to build and exercise the various mechanisms and to then add up the benefits (such as rapid time to market) of each mechanism.

A formula to express the per-product cost of a core asset can be constructed easily:



Minimizing the per-product cost of a core asset is at least a first-order approximation of maximizing the core asset's ROI.

In addition to computing the costs associated with *each* variation mechanism, care must be taken to account for the overall effect of the number of different mechanisms chosen. All other things equal, a small palate of mechanisms is better than a large one. Choosing a small set will involve less training by product developers and contribute to conceptual integrity. These benefits, while hard to quantify, are nevertheless tangible.

In practice, however, it helps to have a way to express the variability requirements imposed by the product set and production strategy. Variability scenarios, discussed in the next section, can help.

5 Variability Scenarios

In order for a product line's variability requirements to be actionable for core asset developers, those requirements must be specified more precisely than just stating, "Build the core assets so they can be adapted for products." As shown by Bass and colleagues, quality attribute requirements can be specified by using quality attribute scenarios [Bass 03]. Scenario-generation tables for a variety of different quality attributes help core asset developers specify quality-attribute-specific scenarios. This same approach can be used to treat variability as a quality attribute for core assets.

Table 1 shows a preliminary scenario-generation table for variability, using the six-part structure described by Bass and colleagues [Bass 03]. The table is not complete, but it can serve as a good starting point for future work.

Table 1: General Scenario-Generation Table for Variability

Scenario Part	**Values**
Source of stimulus	Exercising variability • development organization • integrator • system administrator • end user
Stimulus	Build variants to support variations in • hardware • feature sets • technologies • user interfaces • etc.
Environment	Range of products affected by the scenario, such as • all • a specified subset • those that include feature set *x* • new products

Table 1: General Scenario-Generation Table for Variability (cont.)

Scenario Part	Values
Artifact	Core asset(s) affected, such as • requirements • architecture • component *x* • test suite *y* • project plan *z* • etc.
Response	The variable part should be built in such a way that • the product can be automatically generated • a set of options can be chosen • functionality can be changed • one or more quality attributes can be changed • a new variant can be created
Response measure	With • cost/effort/time for exercising; zero cost when exercised outside the developing organization, almost zero cost when automated • minimal amortized cost[7] of the core asset over *n* number of products

The following three tables show concrete scenarios to illustrate the usage of the general variability scenario-generation table. Table 2 is based on the example described in Section 3. This example states the need for fast creation of reports.

Table 2: Initial Scenario for the Example in Section 3

Scenario Part	Values
Source of stimulus	Product developer
Stimulus	Product under development requires a new report not currently provided by any product.
Environment	During product development
Artifact	Reporting feature of the product
Response	Software to create a new report is produced.
Response measure	Requires no more than two hours to create a variant

[7] The amortized cost of a core asset is the cost of the core asset divided by the number of products, plus the instantiation cost. This calculation gives a core asset developer the ability to compare the cost for creating a core asset against the expected benefits when using the core asset.

Satisfying the stringent two-hour requirement strongly suggests a generator,[8] which is what Salion opted for initially. After realizing that there were additional requirements—namely, that the cost to build the core asset needed to be within limits—Salion took the position reflected in the scenario shown in Table 3.

Table 3: Adjusted Scenario for the Example in Section 3

Scenario Part	**Values**
Source of stimulus	Product developer
Stimulus	Product under development requires a new report not currently provided by any product.
Environment	During product development
Artifact	Reporting feature of the product
Response	Software to create a new report is produced.
Response measure	• Requires no more than one week of effort to create a variant • Requires no more than two months to build the report

All variability scenarios specify some variation that has to be included in a range of products. The scenarios should specify the core asset(s) affected (such as a document or the software), the targeted person who exercises the variability, and the effort or cost to create both the core asset and the product-specific variant.

[8] A comprehensive catalog of variability mechanisms should also include information on how long it takes to exercise a variability mechanism.

6 Variation Mechanisms

As part of the core asset design, the asset developer has to decide what variation mechanism to choose to encapsulate the variable parts and to provide appropriate support for instantiating the variation mechanism. As stated in Section 2, only mechanisms that are aligned with the production strategy can be chosen. For example, if implementing a mechanism requires specific skills (i.e., database design) and those skills are not available to the organization, choosing that mechanism would not be appropriate. (This is true unless someone in the organization decides to compensate for the lack of skills or knowledge by providing training or hiring the required people.) Or, if implementing a mechanism takes two months but the first product must be delivered in one, choosing that mechanism would not be appropriate. In any case, every variation mechanism has a set of properties that help the decision process. Examples of those properties are

- the skill set required to implement the mechanism, such as server or framework programming
- the cost to implement the mechanism, including the cost to learn the skills necessary for implementation
- the cost and time to exercise the mechanism, including the cost of tools required to use the mechanism for a specific product (such as compilers or generators) and the cost of teaching product developers how to use the mechanism
- the targeted group of users that use the mechanism for product-specific adaptation, such as product developer, integrator, system administrator, and end user
- the impact of the variation mechanism on quality, such as possible performance penalties or memory consumption
- the impact on the mechanism's maintainability

Table 4 provides an overview of some important variation mechanisms with their typical properties. Our intention is not to provide a complete list with concrete property values. Rather, our intent is to describe the general type of information required about variation mechanisms in order to make educated decisions on which mechanism to use to solve a specific problem. The concrete form of a mechanism with its properties can be determined only in the context of a concrete product line. Product line organizations should strongly consider producing a catalog of variation mechanisms that includes the ratings and stakeholders rele-

vant to their organization.[9] Lists of possible variation mechanisms, which can inform such a catalog, can be found in the works by Jacobson and colleagues [Jacobson 97] and Anastasopoulos and colleagues [Anastasopoulos 00].

Table 4: Overview of Some Variation Mechanisms and Their Properties

Variation mechanism	Properties to be built into core assets	Properties to be exercised when building products
Inheritance	**Cost:** Medium **Skills:** Object-oriented languages	**Stakeholder:** Product developers **Tools:** Compiler **Cost:** Medium
Component substitution	**Cost:** Medium **Skills:** Interface definitions	**Stakeholder:** Product developer, system administrator **Tools:** Compiler **Cost:** Low
Plug-ins	**Cost:** High **Skills:** Framework programming	**Stakeholder:** End user **Tools:** None **Cost:** Low
Templates	**Cost:** Medium **Skills:** Abstractions	**Stakeholder:** Product developer, system administrator **Tools:** None **Cost:** Medium
Parameters (including text preprocessors)	**Cost:** Medium **Skills:** No special skills required	**Stakeholder:** Product developer, system administrator, end user **Tools:** None **Cost:** Low
Generator	**Cost:** High **Skills:** Generative programming	**Stakeholder:** System administrator, end user **Tools:** Generator **Cost:** Low
Aspects	**Cost:** Medium **Skills:** Aspect-oriented programming	**Stakeholder:** Product developer **Tools:** Aspect-oriented language compiler **Cost:** Medium
Runtime conditionals	**Cost:** Medium **Skills:** No special skills required	**Stakeholder:** None **Tools:** None **Cost:** No development cost; some performance cost
Configurator	**Cost:** Medium **Skills:** No special skills required	**Stakeholder:** Product developer **Tools:** Configurator **Cost**: Low to medium

[9] Table 4 includes core asset developer, product developer, system administrator, and end user, but many applications have more stakeholders (such as subsystem developers, dealers, and installers) whose roles require an understanding of variability.

The high, medium, and low values for cost are relative values. For example, the cost to exercise parameter values (low) is typically substantially less expensive than creating a new class using inheritance (high).

7 Towards True Variability Management

So far, we have presented an overview of product line variability that describes a number of basic concepts (Section 2), establishes the goal for variability (Section 3), identifies the basic inputs to the variation mechanism selection process (Section 4), introduces variability scenarios as a way to express variability requirements (Section 5), and introduces a few variation mechanisms to show what a catalog might look like (Section 6). Our goal was to provide advice about how to select variation mechanisms, and we have outlined a rudimentary version of the approach for selecting those mechanisms:

1. Gather product information and the production strategy to understand the commonalities and variations needed and constraints on product production.
2. Express variability requirements using variability scenarios.
3. Use catalogs of variation mechanisms to choose the ones that satisfy the high-priority scenarios.
4. Use a cost model to formulate the cost of building and exercising variation mechanisms to finalize the choice.

However, this rudimentary approach is lacking in several areas:

- It is largely an asset-by-asset approach that only vaguely takes into account variation mechanism choices that were made previously. If 30 core assets use 1 mechanism, choosing a completely different mechanism for the 31st core asset is unlikely to be a wise choice, but we do not yet understand how to model this.
- We have not treated the issue of managing the dependencies among core assets. Dependencies impose a large burden on the product line organization. For example, loops have to be avoided. In addition, dependency graphs have to be structured so big decisions drive smaller ones (in terms of business impact), not the other way around. Questions regarding who is responsible for tracking dependencies and who is responsible for recognizing, maintaining, and enforcing consistency have to be addressed.
- We have not paid enough attention to the stakeholders for variation mechanisms other than the core asset developers and product developers. In a product line organization, several groups of people are responsible for creating and using core assets. For example, marketing is responsible for the product portfolio and roadmap, developers create components and test cases, product management creates project plans and schedules, technical writers create documentation, and so on. As shown in Figure 2, the different groups have different interests and look at variability from different perspectives, and the infor-

mation exchange between the groups and what variability they considered may not work as effectively as required.

- Our prototypical mechanism catalog in Section 6 is naïve and vague, especially with regard to a mechanism's costs and other factors that would lead to its selection or avoidance.

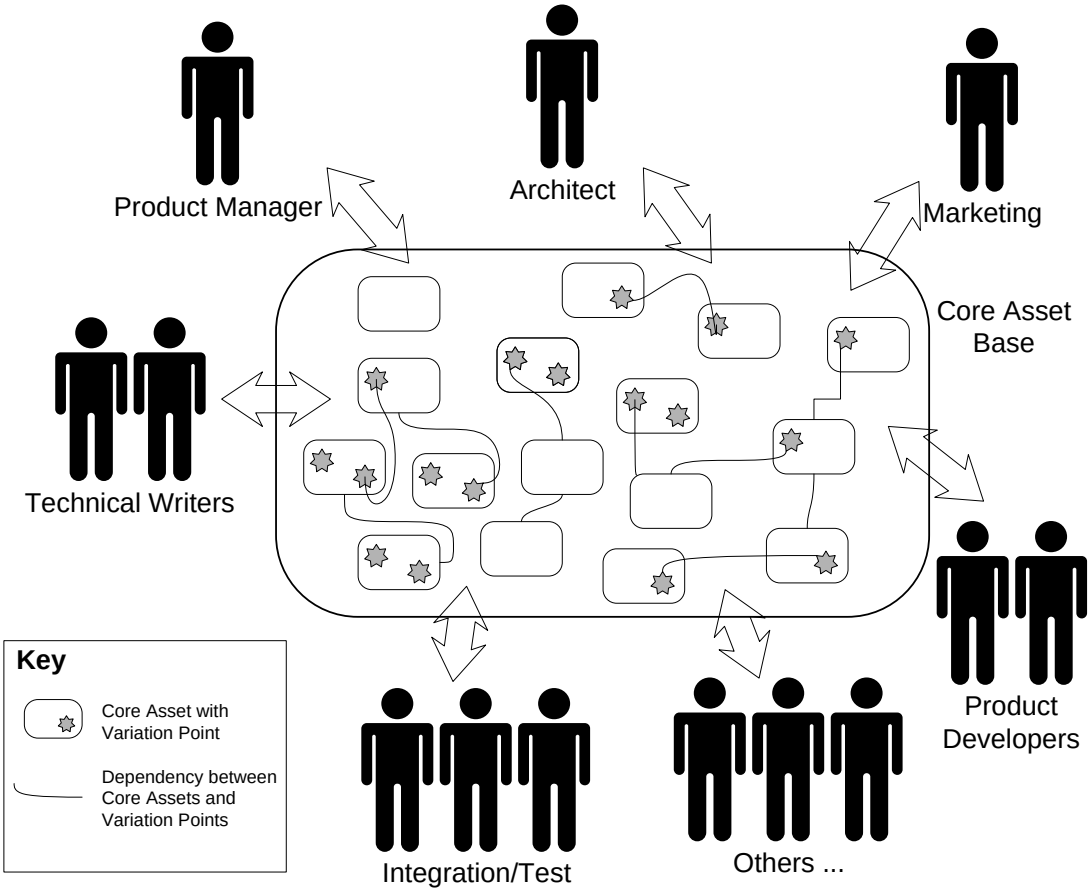


Figure 2: Contributors to the Core Asset Base

Solving these and other problems would enable the systematic management of variability. One could imagine a centralized model that includes information extracted from the core assets, especially the dependency information. This model would enable the product line organization to predict what is involved in building a specific product. It would also enable the organization to control variability throughout the lifetime of the product line and to avoid uncontrolled proliferation of variability that could make product development unnecessarily complicated.

This model could also be used to extract the necessary information to create the production plan for a specific product. As shown in Figure 3, variabilities in core assets influence the

production plan that is used to build products. The production plan contains the description of which variation mechanisms are relevant and how to execute them to build a specific product.

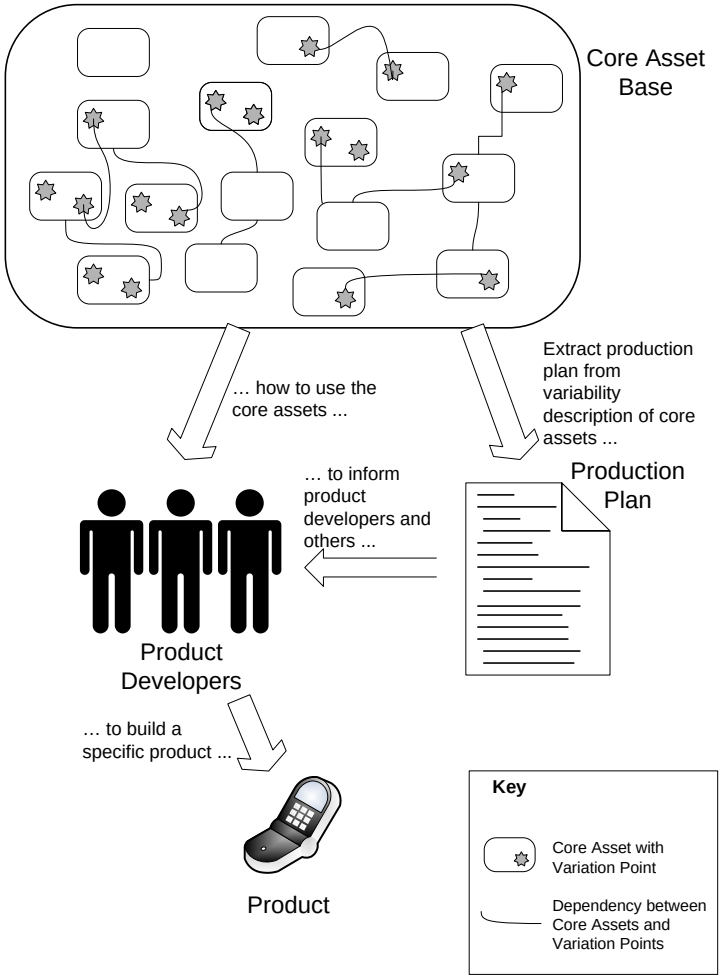


Figure 3: Variabilities in Core Assets Influence the Production Plan

In reality, it may be difficult to convince all but the most mature product line organizations to create such a centralized variability model. Creating such a centralized model looks like a daunting task because the organization would have to get all the necessary information from all core asset developers involved. Even if a centralized model were created, would it be kept current? The changing requirements and the availability of technologies will influence the variabilities.

All creators of core assets should have a clear sense of what is common and what is variable in the asset they create. They should know under which conditions something has to change in the core asset and what variation mechanisms are available to support this change. To profit from this knowledge, an infrastructure—either a technical infrastructure, such as a tool, or an organizational infrastructure, such as an enforced process—has to be provided that extracts this knowledge from the core assets and its developers, no matter what tool was used to

create the asset. The extracted information can then be combined automatically or semi-automatically into a single variability model to capture dependencies and possible inconsistencies.

An organization lacking such an infrastructure will have difficulty predicting the cost to create a specific product from the core asset base. The lack of such an infrastructure also leaves the organization without the means to control variability systematically throughout the lifetime of the product line.

The evolution of the product line can be controlled with the essential variabilities (see Section 2) using the generated variability model, by carefully adjusting the variabilities and clearly communicating them to the creators and maintainers of core assets. Observing the evolution of the essential variabilities over the lifetime of a product line can give some insights into the line's fitness. If the number of essential variabilities increases to an unmanageable number, the scope of the product line might be too broad.

A second way for controlling variabilities is to limit the use of variation mechanisms. Although there might be a perfect variation mechanism for every variable part in all the core assets, it is much easier to just choose from a limited set of mechanisms, even if those mechanisms would be suboptimal. With a small set, it is easier to teach new product developers how to use the mechanisms, the effort for maintaining the mechanisms is lower, and it is easier to automate product derivation. Therefore, when it comes to the ability to manage variability, smaller is better!

Diagrams like the one in Figure 4 can be created from a generated variability model to help check for inconsistencies and to help estimate the cost of certain variability "narratives" (i.e., an initial set of variability decisions leading to others, which in turn lead to others, and so forth).

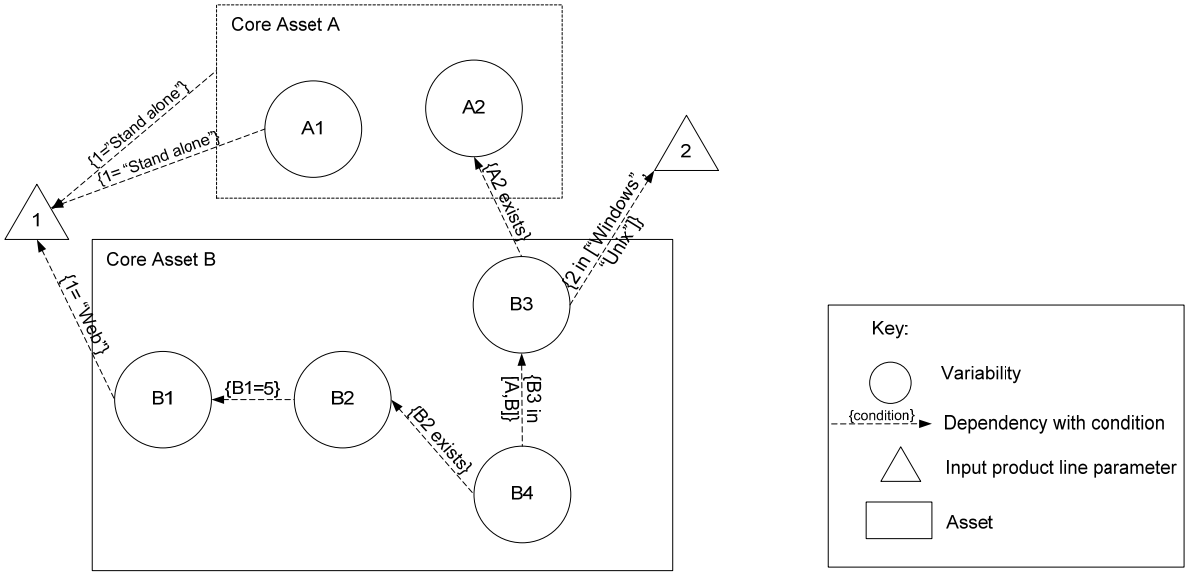


Figure 4: A Variability Model Showing Assets with Their Variabilities and Dependencies

The model in Figure 4 includes all the variabilities of all the core assets, their dependencies with the description of the conditions, the cost of exercising the variation mechanisms for the core assets, and the parameters that must be specified to determine the relevant variabilities.

Here is how to read the diagram in Figure 4. Assume that Input 1 is provided as “Web,” and Input 2 is empty. This assumption leads to the following configuration:

- Core asset A is not included because it should be used only when Input 1 equals “Stand alone.” Therefore, the variable parts A1 and A2 can also be ignored.
- Core asset B is always included. This asset does not depend on anything else. Therefore, all the included variable parts have to be considered.
- Variable part B1 has to be considered because it depends on Input 1 equaling “Web.” Dependent on what the *process* part of this variable part states, either an existing variant can be selected or a new variant has to be created. Assume that an existing variant is selected and the value of B1 is set to 5.
- Variable part B2 has to be considered too because it depends on B1 having the value of 5. Again, assume that a variant can be selected and the value of B2 is set to “xyz.”
- Variable part B3 does not have to be considered, and its value is set to empty because it depends on the existence of A2, which is not included. B3 also depends on Input 2 having the value of “Windows” or “UNIX,” which is not provided.

8 Related Work

There are ongoing activities in the research community, as well as in development organizations, to understand and address the problem of variability in software product lines. Many publications can be found by searching the Internet. We've included a number of references in this report to provide an overview of the current activities in this area. This list is by no means complete; it is meant to be a starting point.

The Reusable Asset Specification (RAS) Consortium is an industry association with the objective of enabling the supply, management, and consumption of reusable software assets. The business objective of the RAS Consortium is to define an industry standard for the identification, management, and consumption of reusable software assets [OMG 04]. The RAS Consortium defines assets as a package of relevant artifacts that provide a solution to a problem. Besides the artifacts, an asset also includes further descriptions, such as an overview, a classification, a solution, and a usage. This description is provided to improve the reusability of an asset.

Thiel and Hein at the Robert Bosch Corporation present a model for product line variability [Thiel 02]. They extend the American National Standards Institute/Institute of Electrical and Electronics Engineers (ANSI/IEEE) 1471 recommended practice for architectural description [IEEE 00] to address the needs for documenting product lines by introducing extensions to model variability in features and in the architecture. Features are organized into a feature model, which describes functional and nonfunctional requirements of the members or the product line. The feature model structures the requirements into a tree that shows common and variable features for the product line variants and a network that places constraints on what combinations are possible. The architectural variability model is described by architectural variation points. Architectural variability represents alternative design options that are not bound during the modeling of the product line architecture.

Jaring and Bosch [Jaring 02] suggest a representation and normalization of variability as a step to a frame of reference. Variation mechanisms are classified and represented according to the introduction and the binding time of variations. They emphasize the need for keeping binding time flexible.

Van Gurp, Bosch, and Svanberg [van Gurp 01] assert that variability can be associated with different "abstraction levels" that a system undergoes in development (from requirement specification to running code). They introduce a terminology for describing variability in terms of variations and variants. The authors focus on features as a useful abstraction for describing variability.

Charles Krueger has characterized and compared different software product line approaches to aid in the selection of the best approach (including choice of variation mechanisms) for different production scenarios [Krueger 03]. He emphasizes the overall maintenance cost (not just the creation cost) of the core assets and products.

9 Summary and Next Steps

This report has laid out some basic concepts related to variability in a software product line (Section 2). The goal of variability is to maximize the ROI of building and maintaining products, and the variation mechanisms contribute to this ROI via (1) the cost to build the core assets using the mechanism and (2) the cost to exercise the mechanism in order to build products (Section 3). A core asset developer, in order to choose wisely from among the many variation mechanisms available, needs to have information at hand (and to search it out if not readily available) about the product set being built and the production strategy in place (Section 4). Variability scenarios are a convenient way to express the variability requirements for a core asset; these scenarios help the core asset developer make an initial choice among variation mechanisms (Section 5). There is a need for more comprehensive variation mechanism catalogs, but core asset developers can use the expression of variability requirements to determine the relevant properties of available mechanisms and use comparisons among those properties to aid in the selection process (Section 6). An overarching goal is the construction (or automated generation) of comprehensive variability models that enable organizations to manage variability in core assets in a unified way, identify dependencies and inconsistencies, and assess the impact of business driver changes and evolution (Section 7).

More work remains before the task of choosing a variation mechanism for each core asset is purely formulaic. Planned work includes

- modeling the cost to build and cost to exercise a variation mechanism using a product line cost-modeling approach such as SIMPLE [Clements 05c]
- expanding the variation mechanism catalog to include more quantitative property descriptions and to ensure that modern product line approaches such as feature-model-based generation approaches [Batory 05] are included
- forming a step-by-step approach for selecting variation mechanisms and validating it with case studies
- continuing to formulate the vision for variability management begun in Section 7

Glossary

configurator: A tool for producing product assets that works by concatenating together blocks of source code (if the core asset is a software component) or documentation sections that are stored in the core asset base as separate files.

core asset: An artifact or resource that is built to be used in the production of more than one product in a software product line. A core asset may be an architecture, a software component, a process model, a plan, a document, or any other useful artifact used in building a system.

essential variability: Variability supporting the ways in which products are required to differ. It is determined by a scoping exercise or requirements analysis for the product line.

local variability: Variability that is provided in a core asset that is not essential variability. An example is making a component variable with respect to which test harness it can work with, for the purpose of simplifying testing.

product asset: An artifact that is part of a product in a software product line. In this report, we include all artifacts that are associated with producing the product, not just the product's deliverable software.

variability: The ability of a system, an asset, or a development environment to support the production of a set of artifacts that differ from each other in a preplanned fashion. In this report, we are concerned with a restricted form of variability, namely, the ability of a core asset to adapt to usages in the different product contexts within the product line scope.

variable part: The part of an asset that is allowed to vary.

variant: The realization of a variable part of a core asset achieved through exercising its variation mechanism(s).

variation: The way in which two or more variants differ from each other. Variation is usually described in general terms, such as the capabilities or properties that the variants have, and not in terms of the exercised variation mechanisms. For example, we might speak of two variants as supporting variation in the communication protocols they support, rather than in the parameter values fed to their embedded #ifdef statements.

variation mechanism: A mechanism to support the creation and/or selection of variants that are compliant with the constraints for a variable part of a core asset.

Appendix: Detailed Explanation of Figure 1

The purpose of this appendix is to provide a detailed explanation of Figure 1, which was presented in Section 2 (page 8) of this report.

The highlighted part of Figure 5 indicates that every core asset consists of one common part and zero or more variable parts. A core asset can be used as is if the asset has no variable part; some adjustments must be made if the asset has one or more variable parts. The purpose of a variable part is to localize the product-specific adjustments to a few places.

Not having a common part in a core asset would mean that everything has to be implemented specifically for the product. This contradicts the nature of a core asset.

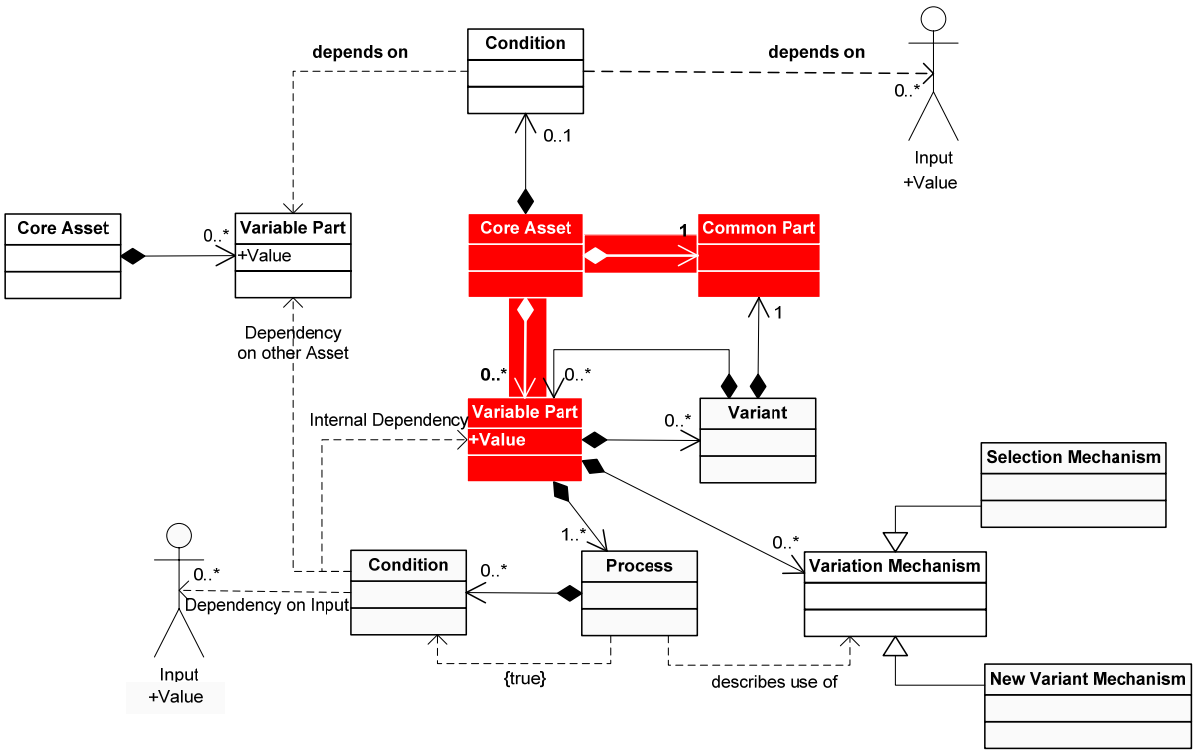


Figure 5: Core Assets Consist of a Common Part and Possibly Multiple Variable Parts

As shown in Figure 6, every variable can have multiple variation mechanisms. The purpose of a variation mechanism is to support the creation or selection of variants. Variation mechanisms can be implemented as part of the core asset itself (such as generic classes from which the code in the variable part can be inherited) or implemented completely outside the asset (such as a code generator that generates variants).

In many cases, a variable part will include at least two mechanisms: one to support the creation of new variants (New Variant Mechanism) and another to support the selection of existing variants (Selection Mechanism).

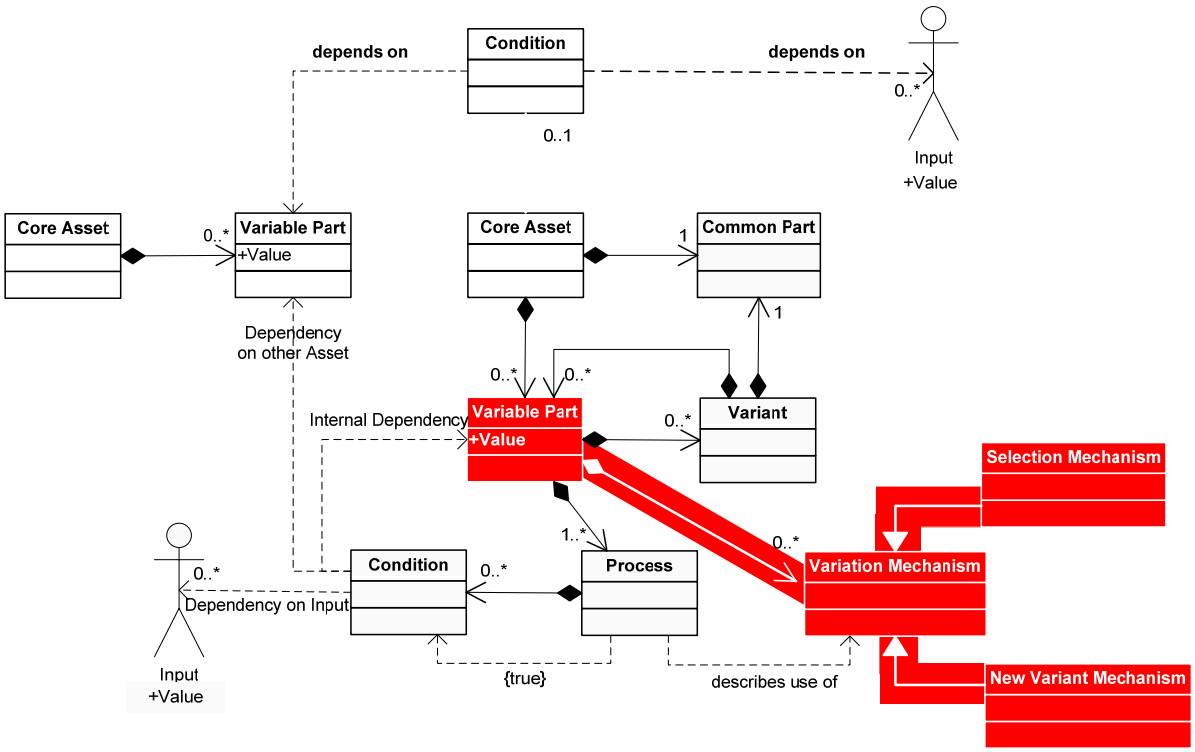


Figure 6: Variation Mechanisms Assigned to a Variable Part

A variation mechanism includes everything provided to product developers so they can make any necessary product-specific adjustments. Items provided to developers include the mechanisms themselves (such as a plug-in mechanism) as well as everything required to use them (such as tools, descriptions, and example implementations).

Although Figure 6 indicates that a variable part may not need to have a variation mechanism assigned (0..* multiplicity), this is not very likely to occur because it would mean that product developers could do what they want here, and the core asset developers would not support them in their task.

As defined in Section 2, a variant is a product-specific realization of a variable part and is produced by exercising the variation mechanism. Typically new variants are created by using the “New Variant Mechanism.” An example of such a mechanism is the use of templates (as

shown in Figure 7). The variation mechanism provides a template that should be used to write a product-specific chapter in a users' guide. It is important to use this template to adhere to formatting rules for the complete users' guide.

After at least one variant is created, it can be included in other products by using the "Selection Mechanism" of the included variation mechanism. Elaborating on the template example, after the product-specific section is written, this section has to be included in the correct place of the complete users' guide. The selection mechanism provided here could be a script that, when executed, ensures that the section is included in the correct place.

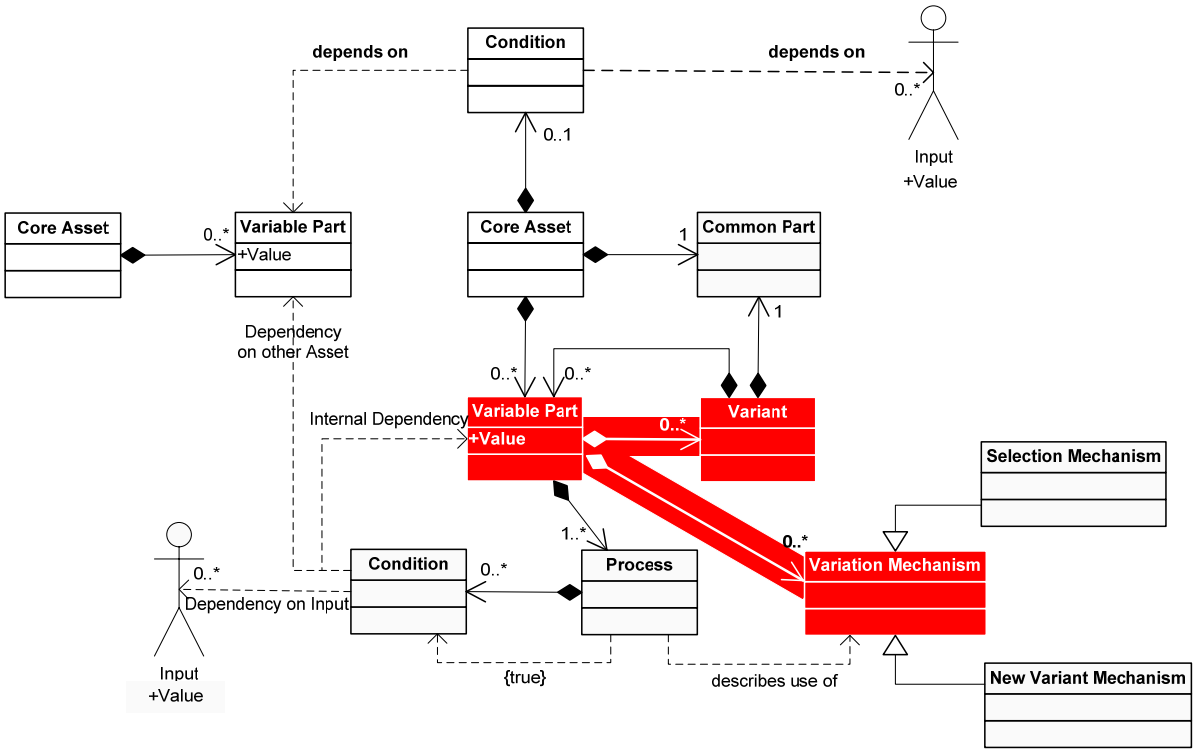


Figure 7: Creating or Selecting Variants

As shown in Figure 8, a variant can also consist of a common part and some variable parts. This type of variant is used if the implementation will be reused in a small subset of products. For example, as shown in Figure 9, a project plan's core assets contain a list of tasks that must be executed to build a specific product. That plan also contains variable parts that depend on specific product requirements. In our example, we assume that products can be created to accommodate different operating systems (OS A and OS B). Therefore, our project plan contains a variable part that depends on what operating system has to be used. Two variant task lists have been created that describe the tasks that have to be performed according to the chosen operating system. One of those variant task lists will be included in the project plan. The variant task lists in our example have a common part that is used as is, but they also have a variant part that describes different tasks for handling the user interface.

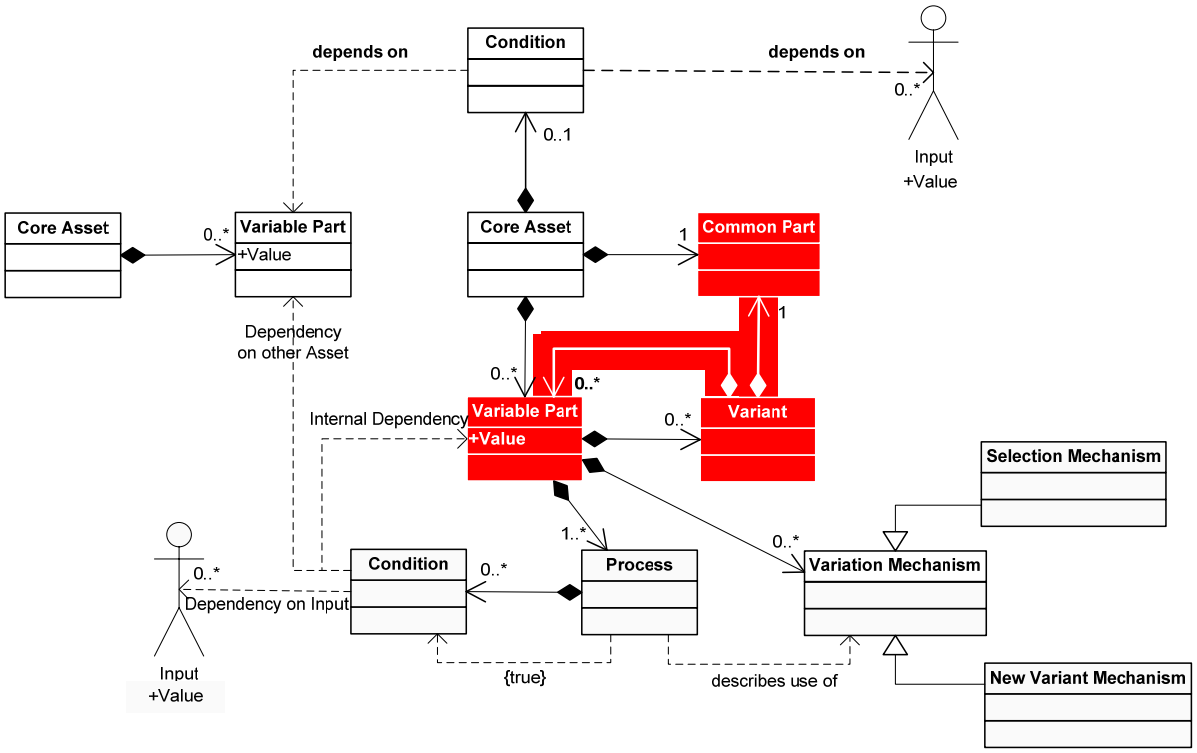


Figure 8: A Variant Can Consist of a Common Part and Variable Parts

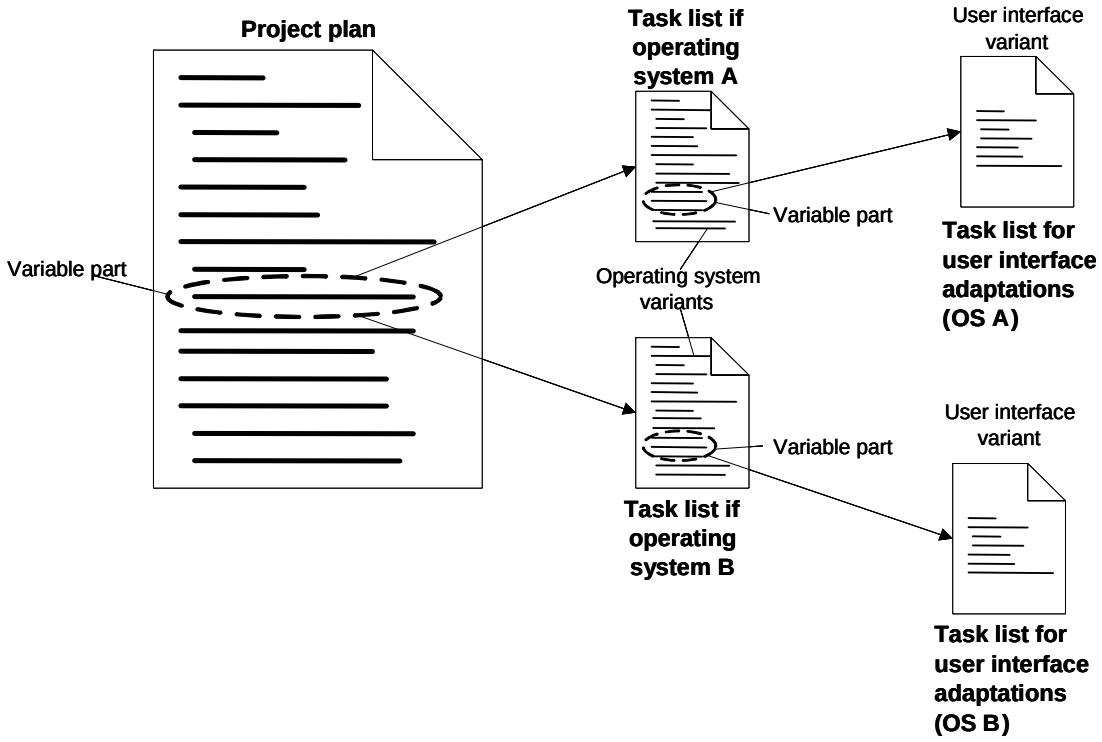


Figure 9: Example of Variants That Have Variable Parts

As shown in Figure 10, every variable part of a core asset has a process description that explains how to exercise the variation mechanism(s) in case the variable part has to be adjusted for a specific product. The process also has a condition attached to determine if the variable part has to be considered. The process combined with the condition is called the “attached processes” of core assets in a product line [Clements 02b].

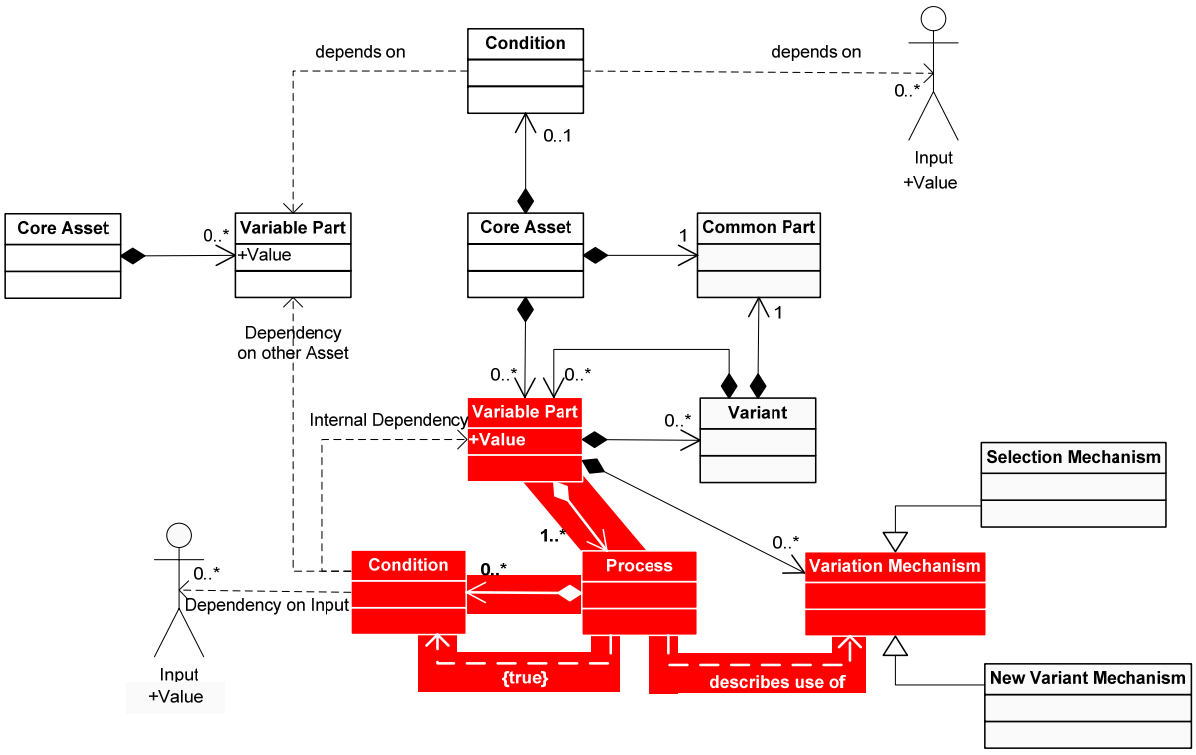


Figure 10: Attached Processes

The highlighted part of Figure 10 reads as follows:

> Execute all the processes attached to a variable part if their assigned condition or conditions are true. The processes then describe how to execute the included variation mechanisms to create or select variants for the product.

A process does not necessarily need to have a condition attached. If there is no condition, the process is unconditional and has to be executed every time a product is built from the core assets.

The condition of a variable part of the core asset is a description of the dependencies on the values that depict a specific product. Values can be either provided by other variable parts or specified by the product developer(s).

Values assigned to variable parts reflect the decisions made for those parts. For example, assume that to build the product a new variant has to be created by exercising the “New Variant Mechanism” (as shown in Figure 11). The value of this variable part would reflect this deci-

sion; for example, it gets “New” assigned. Other conditions can then check this value (such as “variable part A, value = New”) to activate their processes.

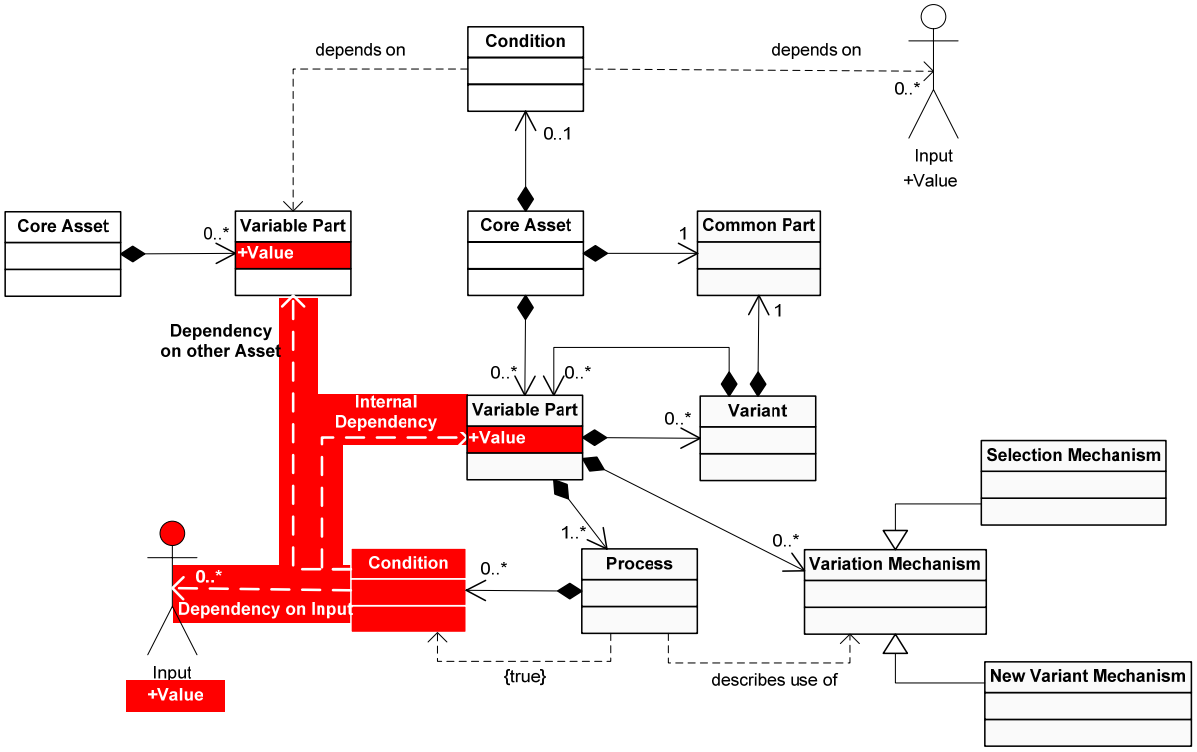


Figure 11: Dependencies on Input or Other Variable Parts

Figure 11 depicts three types of dependencies:

1. A variable part depends on another variable part in the same asset (internal dependency); for example, an architecture design specifies that Component B must be included if Component A is included.
2. A variable part depends on another variable part in a different asset; for example, a users’ guide includes a specific section if a specific feature (as defined in a requirements document) is part of the product.
3. A variable part depends on input from a person; for example, a test plan specifies that a test case should be included in the test suite only if a tester selects it.

So far, we have discussed the dependencies of variable parts and ignored the fact that at least some of the core assets themselves are conditional. As shown in Figure 12, it is very likely that only a subset of the available core assets is used to build a specific product. Therefore, we also need to specify under which condition a core asset is included in a product. If that condition is true, the variable parts of that asset might have to be adapted, depending on their condition.

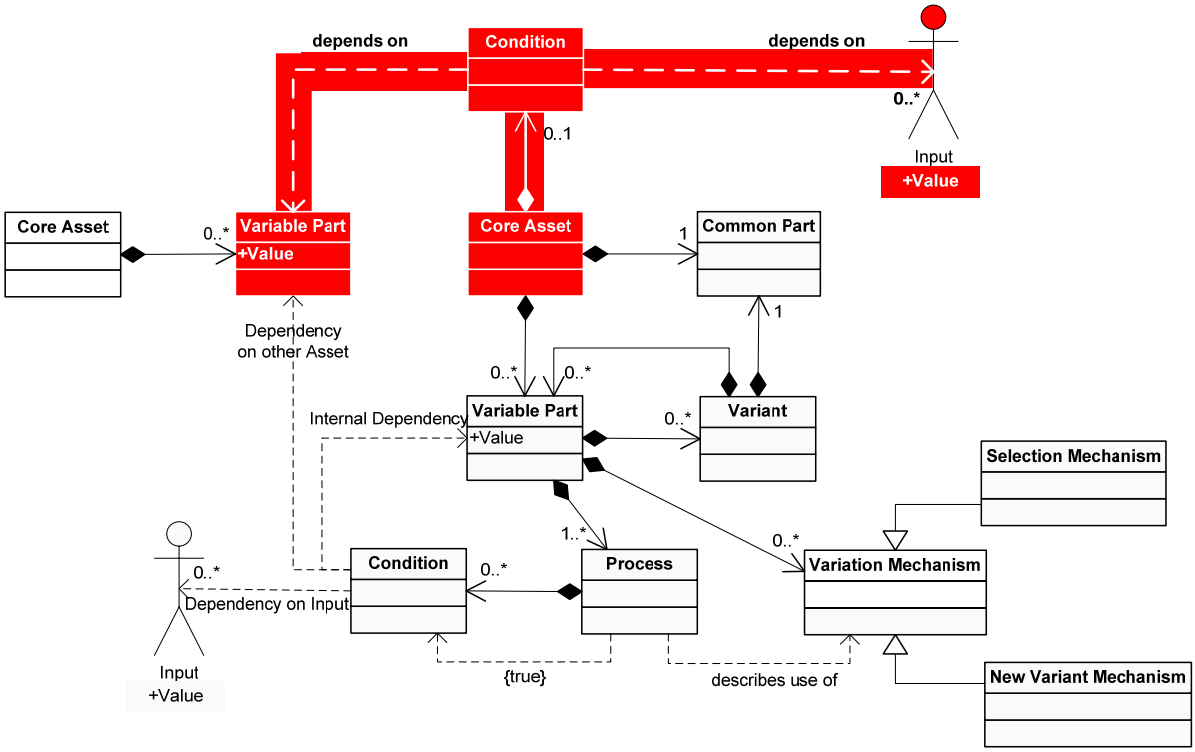


Figure 12: Conditional Core Asset

References

URLs are valid as of the publication date of this document.

[Anastasopoulos 00] Anastasopoulos, M. & Gacek, C. *Implementing Product Line Variabilities* (IESE-Report No. 089.00/E, V1.0). Kaiserslautern, Germany: Fraunhofer Institut Experimentelles Software Engineering, 2000.

[Bass 03] Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice, Second Edition*. Reading, MA: Addison-Wesley, 2003.

[Batory 05] Batory, D. "Feature Models, Grammars, and Propositional Formulas," 7-20. *Proceedings of the 9th International Software Product Lines Conference (SPLC 2005)*. Rennes, France, September 26-29, 2005. New York, NY: Springer, 2005.

[Bosch 00] Bosch, J. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Reading, MA: Addison-Wesley, 2000.

[Chastek 02] Chastek, G. & McGregor, J. *Guidelines for Developing a Product Line Production Plan* (CMU/SEI-2002-TR-006, ADA407772). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. http://www.sei.cmu.edu/publications /documents/02.reports/02tr006.html.

[Clements 02a] Clements, P. & Northrop, L. *Salion, Inc.: A Software Product Line Case Study* (CMU/SEI-2002-TR-038, ADA412311). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. http://www.sei.cmu.edu/publications/documents/02.reports /02tr038.html.

[Clements 02b] Clements, Paul & Northrop, Linda. *Software Product Lines: Practices and Patterns*. Reading, MA: Addison-Wesley, 2002.

[Clements 05a] Clements, P.; Jones, L.; & Northrop, L. "Project Management in a Software Product Line Organization." *IEEE Software 22,* 5 (September/October 2005): 54-62.

[Clements 05b] Clement, Paul & Northrop, Linda. *A Framework for Software Product Line Practice, Version 4.2*. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. http://www.sei.cmu.edu/productlines/framework.html (2005).

[Clements 05c] Clements, P.; McGregor, J.; & Cohen, S. *The Structured Intuitive Model for Product Line Economics (SIMPLE)* (CMU/SEI-2005-TR-003, ADA441881). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005. http://www.sei.cmu.edu/publications/documents/05.reports/05tr003.html.

[Czarnecki 00] Czarnecki, Krysztof & Eisenecker, Ulrich. *Generative Programming: Methods, Tools, and Applications.* Reading, MA: Addison-Wesley, 2000.

[IEEE 00] Institute of Electrical and Electronics Engineers (IEEE). *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems* (IEEE Std 1471-2000). New York, NY: IEEE, 2000.

[Jacobson 97] Jacobson, I.; Griss, M.; & Jonsson, P. *Software Reuse: Architecture, Process, and Organization for Business Success*. Reading, MA: Addison-Wesley Longman, 1997.

[Jaring 02] Jaring, M. & Bosch, J. "Representing Variability in Software Product Lines: A Case Study: 15-36. *Proceedings of the Second Software Product Line Conference (SPLC2)*. San Diego, CA, August 19-22, 2002. Berlin, Germany: Springer, 2002.

[Krueger 03] Krueger, Charles W. "Towards a Taxonomy for Software Product Lines," 323-331. *Proceedings of the 5th International Workshop on Product Family Engineering*. Siena, Italy. November 4-6, 2003. New York, NY: Springer, 2003. http://www.biglever.com/papers/KruegerTaxonomy_PFE5.pdf.

[Krueger 05] Krueger, Charles W. *New Methods Behind the New Generation of Software Product Line Success Stories* (Technical report #200601011). Austin, TX: BigLever Software, 2005.

[Lee 02] Lee, K.; Kang, K. C.; & Lee, J. “Concepts and Guidelines of Feature Modeling for Product Line Software Engineering,” 62-77. *Proceedings of the Seventh International Reuse Conference (ICSR7): Software Reuse: Methods, Techniques, and Tools.* Austin, TX, April 15-19, 2002. New York, NY: Springer-Verlag, 2002.

[McGregor 02] McGregor, John D.; Northrop, Linda M.; Jarrad, Salah; & Pohl, Klaus. “Guest Editor’s Introduction: Initiating Software Product Lines.” *IEEE Software 19*, 4 (July/August 2002): 24-27.

[OMG 04] Object Management Group (OMG). *Reusable Asset Specification* (OMG document ptc/04-06-06). http://www.omg.org/docs /ptc/04-06-06.pdf (2004).

[Thiel 02] Thiel, Steffen & Hein, Andreas. “Modeling and Using Product Line Variability in Automotive Systems.” *IEEE Software 19*, 4 (July/August, 2002): 66-72.

[van Gurp 01] van Gurp, J.; Bosch, J.; & Svahnberg, M. “On the Notion of Variability in Software Product Lines,” 45-55. *Proceedings of The Working IEEE/IFIP Conference on Software Architecture (WICSA 2001)*. Amsterdam, Netherlands, August 28-31, 2001. Los Alamitos, CA: IEEE Computer Society, 2001.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	September 2005	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Variability in Software Product Lines	FA8721-05-C-0003

6. AUTHOR(S)

Felix Bachmann, Paul C. Clements

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2005-TR-012

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	ESC-TR-2005-012

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Product line engineering is a widely used approach for the efficient development of whole portfolios of software products. The basis of the approach is that products are built from a *core asset base*, a collection of artifacts that have been designed specifically for use across the portfolio. To account for differences among the software products, some adaptations of the core assets are usually required. These adaptations should be planned before development and made easy for the product developers to use without jeopardizing existing properties of the core assets.

In a product line with a large number of products and core assets, as well as requirements to make fine-grained adjustments, managing variability can become problematic very quickly. Mismanagement may result in adding unnecessary variability, implementing variation mechanisms more than once, selecting incompatible or awkward variation mechanisms, and missing required variations. As the product line grows and evolves, the need for variability increases, and managing the variability grows increasingly difficult.

This report describes the concepts needed when creating core assets with included variability. These concepts provide guidelines to core asset creators on how to model the variability explicitly, so it is handled consistently throughout the product line and managing the variability becomes feasible.

14. SUBJECT TERMS	15. NUMBER OF PAGES
software product line, product line scenario, variability, variation mechanism, production plan, variation point, core asset	60

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL