The Structured Intuitive Model for Product Line Economics (SIMPLE)

Paul C. Clements
John D. McGregor
Sholom G. Cohen

February 2005

TECHNICAL REPORT
CMU/SEI-2005-TR-003
ESC-TR-2005-003

Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

The Structured Intuitive Model for Product Line Economics (SIMPLE)

CMU/SEI-2005-TR-003
ESC-TR-2005-003

Paul C. Clements
John D. McGregor
Sholom G. Cohen

February 2005

Product Line Practice Initiative

This report was prepared for the

SEI Joint Program Office
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.

FOR THE COMMANDER

Christos Scondras
Chief of Programs, XPK

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 F19628-00-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

This work grew out of a discussion group about product line economics at the 2003 Dagstuhl seminar on product family development. In addition to McGregor and Clements, that group included Dirk Muthig and Klaus Schmid of Fraunhofer IESE and Günter Böckle of Siemens. That group went on to publish two papers about their work: one on the basic model [Böckle 04b] and the other showing how to use the economic model to calculate return on investment in a software product line approach [Böckle 04a]. Although this report is the work of the authors listed, we would not have had anything to write about without the contributions of our German colleagues, with whom we continue to meet and collaborate to further the work. To them we express our gratitude and acknowledge their helpful comments on this report. We are also grateful to Charles Krueger of BigLever Software, Inc., who made many helpful suggestions during recent discussions and to Gary Chastek and Lawrence Jones of the Carnegie Mellon® Software Engineering Institute (SEI) for their helpful reviews. And finally, we thank Dale Peterson of Convergys and John Gaffney of Lockheed Martin for their insightful reviews and comments. Dale Peterson's paper, "Economics of Software Product Lines" [Peterson 04], from which we have quoted liberally in this publication, was especially illuminating.

® Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

Abstract

Software product line practice is an effective strategy for developing families of software-intensive products. Business modeling is a fundamental practice that provides input into a number of decisions that are made by organizations using or considering using the product line strategy. This report presents the Structured Intuitive Model of Product Line Economics (SIMPLE), a general-purpose business model that supports the estimation of the costs and benefits in a product line development organization. The model supports decisions such as whether to use a product line strategy in a specific situation, the specific strategy to apply, and the appropriateness of acquiring or building specific assets. This report illustrates the model's scope by presenting several scenarios and its usefulness by integrating it into several product line practice patterns. The report ends with a description of future work aimed at making the model usable by product line practitioners.

1 Introduction

Product line practice has become an important and widely used approach for the efficient development of complete 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 task. Products are built from a *core asset base*, a collection of artifacts that have been designed specifically for use across the portfolio. This approach has produced order-of-magnitude economic improvements compared to one-at-a-time software system development [Clements 02b]. The product line approach is a comprehensive software-intensive product development strategy. It includes not only technical approaches to solving the problem at hand but business considerations as well, including the economic characteristics of the development. The strategic nature of these characteristics affects how business modeling is carried out in the organization.

The product line approach is not always the best economic choice for developing a family of related systems. For example, the systems may be prohibitively dissimilar from each other, or the family might contain too few systems to recoup the cost of developing the core assets. Decision makers must be able to predict the costs and benefits of developing and evolving a product line as compared to undertaking traditional development approaches. Even when a product line approach has been embraced, alternative approaches are available, and their economic implications must be weighed before one of those approaches is chosen. Peterson writes

> *The investment costs and risks, as well as the downstream benefits can be significant. Gaining executive stakeholder buy-in to making the investment requires a business case that translates the qualitative benefits so often cited for software product lines into concrete business benefits. The business case process can be an effective decision making tool for not only deciding whether an organization should transition to an SPL [software product line], but also how best to make the transition a success. An understanding of the size and timing of the cash flows will enable the organization to assess its transition strategy, and determine the degree to which it should adopt an incremental and iterative approach* [Peterson 04].

Most of the current economic arguments are based on singular data points derived from case studies [Clements 02a, Knauber 02, Clements 02b, Cohen 04, SPL 05a] or convincing arguments appealing to reasonableness and simplistic cost curves [Weiss 99, Cohen 03]. Existing models of development costs in the context of reuse can only be applied in a restricted way, since product line development involves some fundamental assumptions that are not reflected in these models. Currently, only a few economic models exist specifically

for product line engineering, and usually they do not address the effects of maintenance and evolution over time [Favaro 96, Mili 00, Schmid 02, Wiles 02]. Others limit reuse to software only or artifacts close to software, treating cost as a function of lines of code [Gaffney 92]. Most do not take into account the costs of making changes to the organization so that it can more effectively create and sustain a software product line [Cruickshank 93].

A model is needed that can be used to predict software product line costs and benefits under a variety of real-world situations and that can be used easily by product line decision makers not skilled in intricate economic theories. This report suggests such a model called the Structured Intuitive Model for Product Line Economics (SIMPLE), which was developed by the Carnegie Mellon® Software Engineering Institute (SEI).

We have several objectives for SIMPLE:

1. It must model real situations completely and correctly so that it can give high-fidelity answers to the real problems of organizations.
2. It must be sufficiently intuitive for product line personnel to easily produce answers whose derivation can be shown to and understood by others.
3. It should be understandable by managers and technicians alike.
4. It should be flexible enough to help answer a wide range of questions. In fact, our model is structured in parts to allow the modeler to select the appropriate levels of detail and model elements to answer a specific question.
5. It should not assume any particular approach to software product line engineering beyond the basic tenets implied by the definition of a software product line: "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [Clements 02b].

SIMPLE comprises a set of cost and benefit functions that can be used to construct equations that can answer a number of questions such as whether the product line approach is the best option for development and what the return on investment (ROI) is for this approach. In this report, we define these functions and describe some possible implementations for them. We also illustrate the usefulness of these functions by using them to construct the equations for several scenarios.

Of course, even a full-fledged economic model may not provide every input necessary to make the right choice among alternatives. For example, SIMPLE does not address risk—that is, it does not help assess the relative risks of different alternatives and factor them into its formulas.[1] No economic model will replace a manager's experience or instincts or take into account intangibles such as customer loyalty, organizational culture, political influences, and

® Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

[1] For an example of a software cost model that does address risk, see Verhoef's work [Verhoef 04].

personality factors. It will, however, constitute an important tool in a savvy manager's decision-making toolkit.

The remainder of this report is organized as follows. In Section 2, we introduce the complete model and provide specifications for the four basic cost functions at its core. Then, in Section 3, we present scenarios that illustrate the use of the model. These scenarios represent a spectrum of business decisions to which the model can be applied. Section 4 discusses ways of implementing the cost and benefit functions. We provide a general scheme and specific techniques for implementing different portions of the equations. In Section 6, we relate the model to several practice areas in the SEI *Framework for Software Product Line Practice*[SM] [Clements 04], as well as several applicable patterns for software product line practice [Clements 02b]. In Section 6, we sketch out an uncomplicated process for using SIMPLE. In Section 7, we relate the model to previous work in the field. Finally, in Section 8, we lay out future directions for continuing work. Appendix A discusses a homogeneity metric for a family of products, while Appendix B lists and defines the symbols used in SIMPLE.

[SM] Framework for Software Product Line Practice is a service mark of Carnegie Mellon University.

2 Introduction to SIMPLE

In broad terms, SIMPLE can be used to compute estimates for various economic measures related to the building, sustainment, and evolution of software product lines. The problem of cost-benefit calculations for product line engineering that led to SIMPLE can be reduced to the (general) problem of modeling the following situation:

> *An organization has p_init product lines, each comprising a set of products, and s_init stand-alone products. Over a period of time, the organization wishes to transition to the state in which it has p_final product lines, each comprising a (perhaps different) set of products, and s_final stand-alone products. Along the way, the organization intends to add k products or delete d products.*[2]

Figure 1 illustrates this scenario and how it is just one part of an ongoing process.

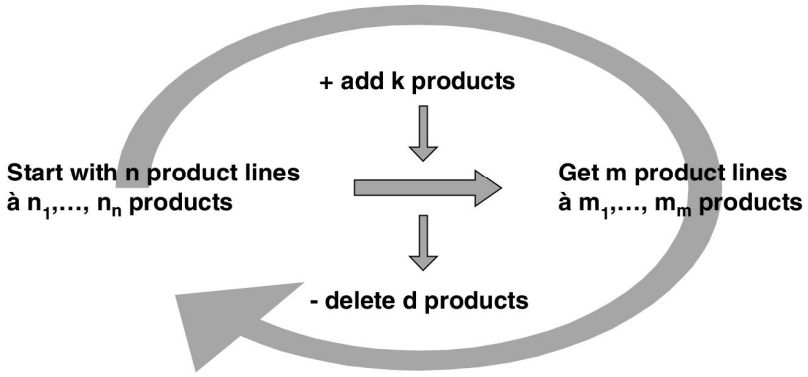


Figure 1: The General Scenario

It is easy to imagine special cases of this scenario that reflect real-world product situations of interest, for example

- An organization has zero product lines and 12 stand-alone products. It wishes to have one product line and zero stand-alone products—that is, it wishes to convert its product collection into a product line. SIMPLE can be used to weigh the cost benefit of doing so, as opposed to letting the products continue to evolve separately.
- An organization has two product lines (perhaps as the result of acquiring another company). It wishes to merge them into one product line while deleting those products that are duplicative. SIMPLE can be used to estimate the cost benefit of merging the product lines, as opposed to keeping them separate.

[2] A dictionary of notation appears in Appendix B.

- An organization has three product lines and wishes to add one product and to know which product line is the best candidate to receive it. SIMPLE can be used to estimate the cost and benefit of adding the product to each of the three product lines, thereby helping to determine the best choice.

In each case (and many others), organizational decision makers will want to know what their plan will cost, what benefits it will bring, and how it compares to other alternatives. As these examples illustrate, SIMPLE can be used to weigh the costs and benefits of one or more product-line-related alternatives, so the most opportune one can be chosen.

Not every scenario that SIMPLE can help with can be couched as a special case of the general scenario described above; however, most of the scenarios we have dealt with to date can be. Section 3 will illustrate the range of scenarios for which SIMPLE is applicable.

2.1 Cost and Benefit Functions

SIMPLE is constructed using a "divide and conquer" approach to the question of how much a particular software product line strategy will cost an organization and how much it gains compared to other alternatives. To express these quantities, SIMPLE introduces *cost functions* and *benefit functions* that describe these constituent aspects of the overall economic question. These functions are introduced beginning in Section 2.2. Rather than rigorously defined mathematical functions, they should be thought of as an invitation to do a thought experiment to come up with a reasonable cost (or monetary benefit) estimate in each area. Each function can be "implemented" through a variety of approaches including the use of an organization's historical data, generalized cost models such as the Constructive Cost Model (COCOMO) II [Boehm 00], or further decomposition into more finely grained functions.

Note: In this report, we may speak of these functions as "returning" a value, but this is meant to invoke the image of an oracle rather than a mathematical subroutine in a programming language. Some readers may be more comfortable substituting "stands for" for "returns."

Section 4 discusses implementation strategies for the cost functions in more detail.

2.2 The Four Basic Cost Functions of SIMPLE

SIMPLE relies on four basic *cost functions*:

1. $C_{org}()$ is a function that, given the relevant parameters, returns how much it costs an organization to adopt the product line approach for its products. Such costs can include reorganization, process improvement, training, and whatever other organizational remedies are necessary.
2. $C_{cab}()$ is a function that, given the relevant parameters, returns how much it costs to develop a core asset base suited to satisfy a particular scope. C_{cab} takes into account the

costs of performing a commonality/variability analysis; defining the product line's scope [Clements 04]; designing and then evaluating a generic (as opposed to one-off) software architecture; and developing the software so designed. C_{cab} also includes building the production plan, establishing the development environment, and producing a testing architecture and other artifacts that are reusable across the family. C_{cab} may be invoked to tell us the cost of developing a core asset base where none currently exists or the cost of deriving a desired core asset base from one or more bases already in place. Note that the core asset base can (and should) also include non-software assets such as plans, schedules, budgets, the scope definition, and various kinds of documentation. It also includes the artifacts that tell you how to produce products from core assets; an example of such an artifact is a production plan [Clements 04].

3. $C_{unique}()$ is a function that, given the relevant parameters, returns how much it costs to develop the unique parts (both software and non-software) of a product that are not based on assets in the core asset base. The result might be a complete product (i.e., one that is not a member of a product line) or the unique part of a product whose remainder is built atop the core asset base of a product line.

4. $C_{reuse}()$ is a function that, given the relevant parameters, returns how much it costs to build a product reusing core assets from a core asset base. C_{reuse} includes the cost of locating a core asset, checking it out of the repository, tailoring it for use in the intended application, and performing the extra tests associated with reusing[3] core assets.

In some product line approaches, every asset is managed as a core asset. An asset such as a piece of software, a plan, or a design specification that only happens to be used in a single product is not treated differently than a corresponding artifact that is used in two or more products. Under this approach, $C_{unique}()$ will simply be zero, and the entire cost of building a product will be carried by $C_{reuse}()$.

A few more cost functions will be added to the model as we go, but these four represent the basic approach.

To show these four cost functions in action, we use them in a very simple expression that describes the cost of building a single product line containing *n* products. This cost can be expressed by Equation 1.

Equation 1

Cost of building a product line =

$$C_{org}() + C_{cab}() + \sum_{i=1}^{n} (C_{unique}(product_i) + C_{reuse}(product_i))$$

[3] John Gaffney suggests the term *multi-use* instead of *reuse* to cover the situation in which material is developed specifically to be incorporated into a set of software products, not just a new version of an existing one. That's a good distinction, but we will continue to employ *reuse* in this report because in most instances we use it as a verb.

This equation says that the cost of fielding a product line is the cost of organizational adoption plus the cost of building the core asset base plus the cost of building each of the *n* products. The cost of building a product is the cost of building the unique part of that product plus the cost of incorporating the core assets into the product.

This equation does not discuss the cost of running or maintaining the product line, nor does it compare the cost with other development methods. These concerns will arise shortly.

The equation is agnostic with respect to whether the core asset base and the products are built entirely from scratch, purchased off the shelf, or salvaged from legacy products. The cost functions will "return" different values depending on the situation.

2.3 C_{prod} : The Cost of Building Products in a Stand-Alone Fashion

A common question to ask about the cost of a product line is this: If we are planning to build *n* products, are we better off building them as a software product line or independently without sharing core assets? Answering this question requires knowing the cost of building them independently. We introduce C_{prod} (*product*) as a cost function that returns the cost of building a product in a stand-alone fashion.[4] This cost function can rely on historical data or general software engineering cost models for its evaluation. The cost of building *n* products independently, then, is expressed in Equation 2.

Equation 2

$$\text{Cost of building } n \text{ stovepipe products} = \sum_{i=1}^{n} C_{prod}(product_i)$$

The cost savings (loss) for building *n* products as a product line versus building them independently is simply [Equation 2] – [Equation 1].

2.4 C_{evo} and C_{cabu} : Modeling the Cost of Evolving a Product

To account for a cycle of product evolution—that is, the time in which a product appears in a new version, probably with new or at least improved features—under the non-product-line regime, the model introduces a new cost function, C_{evo}(). This function is parameterized with product and version numbers and returns the cost of producing that version. Modelers might

[4] This new cost function C_{prod} can be implemented, at least conceptually, by invoking C_{unique} for a product that is 100% unique. In other words, introducing C_{prod} does not make our model any harder to compute and can be thought of merely as a notational convenience.

make a first approximation by assuming that the cost to produce a new version is some percentage of producing the original product; for example

$$C_{evo}() = 20\% * C_{prod}()$$

To calculate the analogous cost under a product line regime, we introduce a new function, $C_{cabu}()$. This function returns a measure of how much it costs to update the core asset base as a result of releasing a new version of a product. Changes to the core asset base can occur because the new version required changes to or exposed bugs in existing core assets. Changes can also occur when new features expose new commonalities with other products that were heretofore considered unique but now can be refactored into commonalities.

2.5 Expressing Time Periods in the Model

Modeling evolutionary cycles has made it clear that SIMPLE must be able to handle time periods or at least have a way to express time periods in which particular events (such as an update) occur. Cost functions need to be interpreted as returning the cost incurred during the period of interest.

For example, up until now, $C_{org}()$ has been applied as though it returns the complete cost of organizational transition. However, managers who are interested in phasing in product lines over time might accomplish the organizational work in stages. Hence, $C_{org}()$ needs to account for the cost during a time period of interest. If everything is done up front, $C_{org}(period\_1)$ accounts for the entire cost, and $C_{org}(t)$, for all other time periods, returns zero.

Similarly, $C_{cab}()$ has to account for the cost of building the core asset base during a period of interest. This is especially relevant under reactive product line development [Clements 02c, Clements 04], which essentially prescribes just-in-time core asset development, as opposed to an all-at-once proactive approach in which the entire core asset base is built up front.

The need to account for time periods is also true of $C_{cabu}()$, the cost of updating the core asset base as the result of producing or evolving a product. Under a reactive product line regime in which the core asset base is constructed from existing or newly fielded products, $C_{cabu}()$ will be high initially but then become lower as the core asset base stabilizes and fills out. Under a proactive regime, the core asset base will stabilize sooner, and $C_{cabu}()$ will stay lower and vary less over time. Under either regime, if the period of interest is one in which products are changing fairly substantially, $C_{cabu}()$ will probably be higher as the core asset base evolves to accommodate them.

The solution, obviously, is to parameterize SIMPLE's cost functions with the period of interest, so their implementation can return values fine-tuned to that period. Adding a period of interest lets the model become much more flexible and carry more fidelity. For example,

one can easily implement the cost functions to take the time cost of money[5] into account, which may, in turn, influence the decision about whether to incur product line expenditures up front or phase them out over time.

In this report, we use *t* to indicate a time period of interest that might be a calendar period or correspond to a product release cycle or milestone. Another time aspect that is useful to model has to do with the binding times that apply to the built-in variabilities. The cost of providing and exercising variabilities is a major factor in deciding which variability mechanisms to choose. SIMPLE can be used to help make these choices, by making the time periods of interest correspond to time intervals bounded by applicable binding times such as those listed on the Web [SPL 05b].

2.6 Accounting for Product Line Economic Benefits

Software product lines bestow benefits to the developing organization besides direct cost savings. For example, they often allow an organization to bring a product to market much more quickly. We can accommodate these other factors by using benefit functions that are similar to the cost functions introduced in the basic model. Unlike the cost functions, there is no fixed number of benefit functions. One product line may be intended to (1) reduce the time required to get products to market and (2) increase productivity. Another product line may only be expected to increase customer satisfaction. Still another may be intended to increase sales by capturing a market share.

We add benefit functions to the model as a variable-sized collection as shown in Equation 3.

Equation 3

$$\text{Benefits achieved from product line approach} = \sum_{j=1}^{nbrBenefits} B_{ben_j}(t)$$

where ben_j is a specific benefit and B_{ben_j} is the benefit function for that benefit. Each benefit function is parameterized by the time period of interest since the benefits may vary over time.

Benefit functions return the economic value of achieving that benefit during the time period indicated. For example, achieving high customer satisfaction during a roll-out period may result in increased sales during a product line's formative period, which brings a quantitative economic benefit.

The contributions of the benefits are summed and then used to build a model equation as needed. For example, recall that to express the development cost savings (or loss) from using the product line approach as opposed to one-off development for each product, we wrote

[5] This refers to the cost of holding onto money—that is, lost investment returns and interest.

[Equation 2] – [Equation 1]. A more complete picture of the cost benefit of using a product line approach adds Equation 3 to that result.

The economic forecaster who is building equations from these functions must take care not to count some benefits twice. First, many benefits result in changes to the four cost functions and hence are accounted for there. For example, increased productivity is often cited as a benefit of product line engineering, but increased productivity is essentially a reduced cost that will already be reflected in the cost of building the core asset base and in the sum of the cost of reuse and the unique portion of the product. To then add a benefit function measuring savings due to increased productivity would result in a model that overstates the savings. Second, many different benefits might lead to the same cost savings. For example, increased productivity also might result in shorter time to market, which, in turn, might result in higher customer satisfaction. But tighter product alignment or higher product quality might also lead to increased customer satisfaction. All these things might lead to increased sales and income. It would be very easy to count a benefit whose cost savings has already been accounted for under another benefit. Modeler beware.

In Section 4.7, we provide some examples of how the benefit functions might be implemented. For now, benefit functions can be thought of as providing an answer to the question "How much would achieving this benefit during this time period be worth to your organization?"

In a cost-benefit analysis, the most visible benefit is the income derived from the sale of the products. If the modeler is concentrating on the costs of software development, the income figure may need to be adjusted based on the portion of the product provided by software. The model should reflect when this income is generated. In Section 2.5, we discuss how to do this by making the formulas time-period specific.

Peterson provides a strong list of benefits (several of which are not often cited) in describing why his company (Convergys) chose the software product line approach [Peterson 04]. His list may serve as a good starting point for producing a useful set of benefit functions:

- **research and development (R&D) investment:** the ability to leverage the R&D investment across the product family by reusing a common set of components
- **subject matter expertise:** the ability to leverage subject matter experts across the product family by concentrating domain subject matter experts with similar skills and knowledge into centralized groups that serve all products
- **productivity and quality:** improving productivity and quality by breaking large, monolithic applications into smaller, more manageable projects and by using components that encapsulate the applications' functionality
- **time to market:** increasing the rate of delivery of new capabilities to market and enabling new products to be delivered faster by reusing well-established components

- **people mobility:** providing employees with more career development opportunities by standardizing the development environment and processes, thereby reducing the learning curve associated with a move to a new project
- **supplier relationships:** standardizing the platforms and development environment enables a more effective leverage of supplier relationships
- **geographic flexibility:** standardizing component-based development facilitates the distribution of development responsibilities across locations
- **sourcing flexibility:** using a modular architecture to enable greater flexibility to build, license, or acquire software
- **product refresh:** using a modular architecture to facilitate the process of refreshing the product family as new technologies and/or software components become available

3 Scenarios

With SIMPLE now fully introduced, this section will give examples of using it to solve a set of scenarios that represent real-life situations in product line organizations.

3.1 The Cost of Building a Software Product Line

Scenario: An organization wishes know the cost of producing a set of products as a software product line.

The cost can be expressed by Equation 4:

Equation 4

Cost of building a product line =

$$C_{org}(t) + C_{cab}(t) + \sum_{i=1}^{n}(C_{unique}(product_i, t) + C_{reuse}(product_i, t))$$

The time parameter *t* refers to the entire time period during which the core asset base and the first releases of each product are constructed. As noted in Section 2, this equation does not take into account any evolution.

3.2 The Cost of Building a Software Product Line Vs. Building the Products Independently

Scenario: An organization wishes to choose between building a set of products as a software product line and building them as a set of stand-alone products that do not share core assets.

The cost savings (loss) for building *n* products as a product line versus building them as stand-alone products is simply [Equation 2] – [Equation 1]:

Equation 5

Cost savings of building a product line vs. stand-alone products =

$$\sum_{i=1}^{n} C_{prod}(product_i, t) \; -$$

$$(C_{org}(t) + C_{cab}(t) + \sum_{i=1}^{n}(C_{unique}(product_i, t) + C_{reuse}(product_i, t)))$$

Example 1: Suppose there are five products, all roughly the same size and complexity, and each one costs about three person-years (PY) to build as a stand-alone project. Since $C_{prod}()$ = 3PY, Equation 2 returns 15PY as the cost of building the family separately.

Suppose it takes about 2PY to build the core asset base for these five products:

$$C_{cab}() = 2PY$$

And suppose it takes about 1PY to turn the organization around:

$$C_{org}() = 1PY$$

Suppose further that each product is about one-half core assets and one-half unique content and therefore, $C_{unique}()$ = 50% * 3PY = 1.5PY. Because reuse literature[6] suggests that using reusable software can cost around 15% of building it, $C_{reuse}()$ = 15% of one-half of 3PY, or 0.225PY.[7] Hence, Equation 1 returns 1PY + 2PY + 5(1.5PY + .225PY) = 11.625PY.

The cost savings of the product line approach in this case, then, is 15PY – 11.625PY = 3.375PY. This savings is calculated over a single time period (the period of building the products), and the scenario does not take into account any changes to the products or the core assets over time.

3.3 Cost of Releasing a New Version of a Product Line Member

Scenario: An organization wishes to know the cost of releasing a new version of a product in a software product line that is already in existence.

The cost of releasing a new version of a product in a product line is given by Equation 6 (which assumes that all the organizational and original core asset creation costs have already been borne):

Equation 6

Cost of releasing a new version of a product in an already-existing product line =

$$C_{cabu}() + C_{unique}() + C_{reuse}()$$

where all terms are suitably parameterized for two purposes: (1) to designate the specific product and version of interest, as well as the time period over which the update is to be performed and (2) to note that the version is an update and not a new development. This equation says that the cost of an update is the cost of updating the core asset base plus the

[6] This value is the accepted value for opportunistic reuse. A product line will have a smaller value than 15% because there is no search or qualification cost. Boehm's COPLIMO model uses a figure of around 5%. We use 15% as a very conservative estimate in this example but suggest that each company measure its local cost.

[7] Notice that C_{unique} and C_{reuse} are both functions of the fraction of each product that the core asset base provides. As that fraction rises, the former decreases and the latter increases.

cost of the unique part of the change plus the cost of using the updated core asset base to make the change.

If all products in a product line are updated in lockstep, this equation can be summed over the set of products to calculate the total cost of an evolutionary update.

Example 2: Suppose each time a product from the Example 1 product line is updated, about 20% of it is new. Suppose also that under a product line regime, about 5% of the core asset base (which, recall, represents a 2PY investment) is updated as the result of a product update. Suppose that since the core asset base accounts for roughly half of the product (as postulated in Example 1), it will account for roughly one half of the new part of the product as well. So the update to our product (which consists of an addition that's about 20% the size of the original) consists of half new code and half core asset code. Therefore

- $C_{cabu}() = .05 * C_{cab}()$
- $C_{cab}() = 2PY$
- $C_{unique}() =$ 1.5PY for an entire product, and so 10% * 1.5PY for an update
- $C_{reuse}() = .225PY$ for an entire product, and so 10% * .225PY for an update

So Equation 6 returns

$$.05*2PY + 10\%*1.5PY + 10\% *.225PY = 0.2725PY$$

This compares favorably with $C_{evo}() = 20\% * C_{prod}() = 20\%*3PY = 0.6PY$, which is what a product update under the stand-alone regime would take.

However, Equation 6 does not take into account the cost of setting up the product line in order to be able to perform inexpensive updates. The next section will deal with that.

3.4 Comparing the Cost of Converting to a Product Line Vs. Continuing to Evolve an Existing Set of Stand-Alone Products

Scenario: An organization has a set of existing stand-alone products undergoing periodic evolutionary updates. Its managers might wish to know which is cheaper:

- ***Option A: converting them to a product line and continuing their evolution in that form***
- ***Option B: continuing to evolve them separately and foregoing the cost of setting up the product line***

The cost of option A for one evolutionary cycle is the cost of setting up the product line plus the cost of evolving the products once using the core asset base. In other words, the cost of option A is Equation 4 plus Equation 6 summed over the products in the product line:

Equation 7

Cost of setting up a product line and evolving it through one cycle =

$$C_{org}() + C_{cab}() + \sum_{i=1}^{n}(C_{unique}(product_i) + C_{reuse}(product_i)) \; + \; \sum_{i=1}^{n}(C_{cabu}(product_i) + C_{unique}(product_i) + C_{reuse}(product_i))$$

However, when we set up the core asset base and reengineer the products for this example, we could take a shortcut. We could aim to produce core assets and reengineer the products to support their next evolutionary cycle. Under this option, Equation 6 drops out of the picture, since the set-up cost given by Equation 4 will aim for the next step in the products' evolutionary trajectory. So we simply have

Equation 8

Cost of setting up a product line that aims for the first evolutionary cycle =

$$C_{org}() + C_{cab}() + \sum_{i=1}^{n}(C_{unique}(product_i) + C_{reuse}(product_i))$$

The cost of option B for one evolutionary cycle is simply $C_{evo}()$ summed over each product in the portfolio:

Equation 9

Cost of evolving a group of stand-alone products through one cycle = $\sum_{i=1}^{n} C_{evo}(product_i)$

The cost savings (if any) of turning the products into a product line and taking them through one evolutionary cycle is Equation 9 – Equation 8:

Equation 10

Cost savings of setting up and evolving a product line once
vs. evolving group of stand-alone products once =

$$\sum_{i=1}^{n} C_{evo}(product_i) \; - [\, C_{org}() + C_{cab}() + \sum_{i=1}^{n}(C_{unique}(product_i) + C_{reuse}(product_i)) \,]$$

The cost savings for moving from the first evolutionary step to the second can be expressed similarly: simply as Equation 9 – Equation 6 summed over all products, because at this point the product line will have already been set up:

Equation 11

Cost savings of second evolutionary step with a product line =

$$\sum_{i=1}^{n} C_{evo}(product_i) -$$

$$\sum_{i=1}^{n} (C_{cabu}(product_i) + C_{unique}(product_i) + C_{reuse}(product_i))$$

Here, the functions are invoked to return the costs associated with product updates, not whole products.

The same equation expresses the cost savings when moving from the *p*th step to the (*p*+1)st step, as long as *p*>1 and the product line has already been set up. To add up the cost savings of setting up the product line and evolving it over *p* rounds, add Equation 10 to the sum of Equation 11 for each of the next (p-1) rounds.[8]

Example 3: Suppose that p is 5. Then the cost savings are equal to

$$\text{Equation 10} + \sum_{t=1}^{4} \text{Equation 11} \quad \textbf{or}$$

$$\sum_{i=1}^{n} C_{evo}(product_i) - [C_{org}() + C_{cab}() + \sum_{i=1}^{n} (C_{unique}(product_i) + C_{reuse}(product_i))] +$$

$$\sum_{i=1}^{4} (\sum_{i=1}^{n} C_{evo}(product_i) -$$

$$\sum_{i=1}^{n} (C_{cabu}(product_i) + C_{unique}(product_i) + C_{reuse}(product_i))$$

Substituting the previous values gives

$$\sum_{i=1}^{5} 3PY - [1Py + 2PY + \sum_{i=1}^{5} (1.5PY + .225PY)] + \sum_{i=1}^{4} (\sum_{i=1}^{5} .6 -$$

$$\sum_{i=1}^{5} (.05) * 2PY + 0.15 + .0225)) = 9.925\text{PY}$$

[8] If the costs of each round are the same, Equation 8 can simply be multiplied by (p-1).

3.5 Return on Investment (ROI)[9]

Scenario: An organization wishes to know the ROI achieved by setting up a software product line and using it as the basis for product evolution. It wishes to know the ROI after the first round of evolution and after subsequent rounds.

Many organizations wish to understand the costs of product line adoption and sustainment in terms of the ROI, which is usually defined as

ROI = *cost savings / cost of investment*

The investment cost of a product line is the organizational cost plus the cost of establishing the core asset base: $C_{org}() + C_{cab}()$.

The cost savings depends on the specific scenario. Suppose we continue the example from the previous section: An organization wishes to choose between continuing to evolve a set of stand-alone products and converting them to a product line. The ROI after one round of updates is Equation 10 divided by the investment cost:

Equation 12

$$\text{ROI achieved after one round of evolution of a product line} = \frac{\sum_{i=1}^{n} C_{evo}(product_i) - [\,C_{org}() + C_{cab}() + \sum_{i=1}^{n}(C_{unique}(product_i) + C_{reuse}(product_i))\,]}{C_{org}() + C_{cab}()}$$

The ROI after p rounds of updates (p>1) is Equation 10 plus (p-1) iterations of Equation 11, all divided by $C_{org}() + C_{cab}()$.

Example 4: Continuing Example 3 from the previous section, we see that the investment cost of $C_{org}() + C_{cab}()$ is 3PY, so the ROI through initiation and four rounds of updates is 9.925 / 3 = 331%.

3.6 Constructing and Evolving a Product Line

Scenario: An organization has zero product lines and wishes to produce s1 products. The organization wishes to know the cost savings (if any) that it will accrue over "nbr_periods" time periods of constructing the s1 products using a product line versus constructing and evolving them in a stand-alone fashion.

[9] Several financial measures are used for investment decisions such as net present value (NPV) and payback period. Any of them can be computed using the information from the cost and benefit functions. In this report, we illustrate that using a simple ROI computation.

There are two ways to prosecute this scenario. The first is by translating "nbr_periods" in the scenario to a number *p* of evolutionary updates and then using Equations 7 and 8 as discussed previously—namely, adding the result of Equation 7 to (*p*-1) times the result of applying Equation 8.

The second way is to use the cost functions parameterized with the time period of interest and interpreted accordingly. Setting up and evolving a product line over "nbr_periods" units of time can be expressed as follows:

Equation 13

Cost of building and evolving product line over "nbr_periods" time periods =

$$\sum_{t=1}^{nbr\_periods}\left[C_{org}(t)+C_{cabu}(t)+\sum_{i=1}^{n_j}C_{unique}(p_i,t)+\sum_{i=1}^{n_j}C_{reuse}(p_i,t)\right]$$

Equation 13 requires the modeler to determine the following in each period: how much organizational cost was incurred, how much the core asset base was updated,[10] how much product-unique development cost was incurred, and how much reuse cost was incurred. The last two terms return evolution costs for those time periods in which evolution is occurring.

Next, we compute the cost of building and evolving the s1 products in a stand-alone fashion (using Equation 2) and sum that over the same time periods:

Equation 14

$$\text{One-at-a-time cost} = \sum_{t=1}^{nbr\_periods}\left[\sum_{i=1}^{s1}C_{prod}(product_i,t)\right]$$

This scenario is "solved" by taking the difference of these two expressions.

Using the time periods allows modelers to experiment with different schemes for building the core asset base and (for example) to see how early they can achieve a positive cost savings.

3.7 Redistributing Products Among Existing Product Lines

Scenario: An organization has n product lines, each comprising a set of products, and also has s1 stand-alone products. The organization wishes to transition to the state in which it has m product lines, each comprising a (perhaps different) set of products, and also s2 stand-alone products. The organization wishes to determine the optimum division of products among the optimum number of product lines to minimize the cost of initial construction and maintenance for the expected life of five years.

[10] As mentioned previously, most of the core asset base could have been built up front or phased in over time.

This is an optimization problem that requires a search of the possible solution space (i.e., all possible allocations of products to product lines) to identify the optimal value. Each point in the search space has a specific number of product lines, each with a specific set of products and a specific number of stand-alone products. As the collection of products in each product line changes, the amount of commonality or homogeneity also changes. This, in turn, changes the value returned by C_{cab} (since the core asset base is responsible for providing that commonality) and the sums of the C_{unique} and C_{reuse} values. The difference among product lines is the size of the core asset base relative to the size of the unique parts of the products.

The cost of each possible solution for this scenario is given by

Equation 15

$$COST(npl, n_j, s_x) = \sum_{t=1}^{numberOfPeriods} \left[\sum_{j=1}^{npl} (C_{org}(t) + C_{cabu}(t) + \sum_{i=1}^{nj} (C_{unique}(p_i, t) + C_{reuse}(p_i, t))) + \sum_{i=1}^{s_2} C_{prod}(t) \right]$$

where

- S_x is the number of stand-alone products and varies over all values of s_1+ sum(n_j) products.
- n_j = number of products in the *j*th product line.
- npl, the number of product lines, is between 0 and s_1+ sum(n_j) products.
- numberOfPeriods = 5.

The solution for the scenario is to find the m and m_j for which cost(m, m_j, s_x) is minimized. That defines a large but finite search space. Future work will explore how a homogeneity metric—a measure of how much alike the products in a product line are—would help narrow this search space.

3.8 Adding New Products to Existing Product Lines

Scenario: An organization has n product lines, each comprising a set of products, and also has s1 stand-alone products. The organization intends to add k products. It wishes to determine the optimum allocation of the k products over the existing product lines based on foreseeable maintenance costs over the next "nbr_periods" time periods.

This scenario can be solved by adding a product to the "best" product line, one at a time. To compute the "best" product line to which to assign a stand-alone product, we compute the cost of adding that product to each product line and the cost of maintaining that product line with the new product added. The product line with the lowest costs in these two areas is the "best."

The cost of adding a product to a product line (assuming this is done in a single time period t) is

Equation 16

$$C_{org}(t) + C_{cab}(t, PL, product) + C_{reuse}(t, PL, product) + C_{unique}(t, PL, product)$$

where the PL parameter indicates the candidate product line to which we're adding the product. This parameterization reminds us to calculate the cost functions differently based on the different product lines.

The cost of maintaining that product line with the new product in it over "nbr_periods" time periods is the equation from scenario #2 above.

Repeating the application of Equation 16 and Equation 2 for each of the k products produces the optimum allocation of products to product lines.

In practice, however, order might matter. That is, putting the first three products into product line #2 might result in product line #2 being much better positioned to accept product #4. However, if we had started with product #4, it might have been assigned to product line #1. In theory, we can address this problem by calculating the cost for all possible assignments of the k products in all possible orders and come up with the least costs. In practice, this will be untenable, but we will probably want to test merely a few specific allocation hypotheses, rather than every possible one.

3.9 Build Vs. Buy

Scenario: An organization has n product lines, each comprising a set of products, and also has s1 stand-alone products. The organization intends to add k products. It wishes to determine the optimum split between building and buying the additional assets required for the k products. The products will be built in the first year and maintained for four additional years.

This scenario builds on the scenario in Section 3.7. By adding k products, the number of product lines, the number of products in each product line, and the number of stand-alone products may change. Once the optimal arrangement has been determined as in the fourth scenario, the newly needed assets are identified. The problem now becomes how to optimize the cost of the new assets.

The driving cost is $C_{cab}(t)$. In most scenarios, $C_{cab}(t)$ returns the original cost of all the assets. In the case where lease or royalty payments must be made and $t > 1$, $C_{cab}(t)$ returns the additional cost for that year's payments. In the case where assets are created in-house and $t > 1$, $C_{cab}(t)$ returns the amount anticipated for maintenance.

The solution for the scenario is the combination of new assets that requires the smallest expenditure. This solution is below the level of visibility of the product line equation and would be handled in the implementation of $C_{cab}(t)$. Section 4 presents some ideas on implementing the cost functions.

4 Approaches for Implementing Cost and Benefit Functions

The name of our model, SIMPLE, indicates two attributes for which we are striving: structured and intuitive. We intentionally defined the model in terms of a set of cost and benefit functions to allow users to provide layers of implementations. This structure allows the modeler to drive the model down to the level of detail for which sufficiently accurate data can be provided. This structure allows the model to be intuitive to a wide range of people by allowing them to view the level in the model that is most useful to them. In this section, we describe some ways of implementing the functions described in Section 2.2.

Modelers begin their work with different levels of information. Some will have data from previous non-product-line projects, some will have data from previous product line projects, and some will have no previous data at all. SIMPLE allows different approaches based on the knowledge at hand.

4.1 Using an Organization's Own Historical Data

The simplest case of function implementation is when the user has estimates for each cost. Some of the data can come from historical information, such as the average cost of building a one-off product. As an example, the four cost functions of Equation 1 are replaced with cost estimates (CE) in Equation 17:

Equation 17

$$CE_{org} + CE_{cab} + \sum_{i=1}^{n} (CE_{unique}(product_i) + CE_{reuse}(product_i))$$

Using historical data is the simplest approach but perhaps not the easiest. The modeler must have extensive quantified experience with the organization's development processes and be able to predict future costs based on past experience. That may not always be possible if the future products are dissimilar to past products.

4.2 Community Benchmarks

The product line literature is slowly building an intuition about many of the costs. These values can be used as the starting point for organizations that have no recorded development

history of their own. For example, the cost of making a component that is reusable is widely estimated to be 150% of the cost of building the component for a single use, whereas the cost of reusing reusable software is thought to be less than 5% of the cost of building it new (if you do not have to go searching for it) [Boehm 04]. Gaffney and Cruickshank show how to use estimated lines of code and industry-standard effort benchmarks to produce cost estimates [Gaffney 92].

Boehm has revised the cost drivers in COCOMO to define a new set for the Constructive Product Line Investment Model (COPLIMO) [Boehm 04]. Considering the cost of building products from scratch in a product line, COPLIMO data shows that between two and three products is the breakeven point, as supported by much of the product line literature [Clements 02b]. COPLIMO will be discussed in Section 7.2.

4.3 Utility Functions

Utility functions can be used to help estimate a cost over each of a number of consecutive time periods. An action or asset has a cost (or value). That cost may be incurred all at once or over time. A modeler may assign a value to each of a set of outcomes. The utility function has a minimum, usually 0, and a maximum that is often taken to be 1 or 100%. A utility function, u(t), returns the amount of the utility (cost or value) for a time period t. For example, if the expected cost of an asset is c dollars, to be paid in four equal installments over four time periods, the utility curve looks like the one shown in Figure 2, and u(t) = 1/4 for t=1, 2, 3, or 4. The expected cost is computed by: c*u(t) or c/4 for each period.

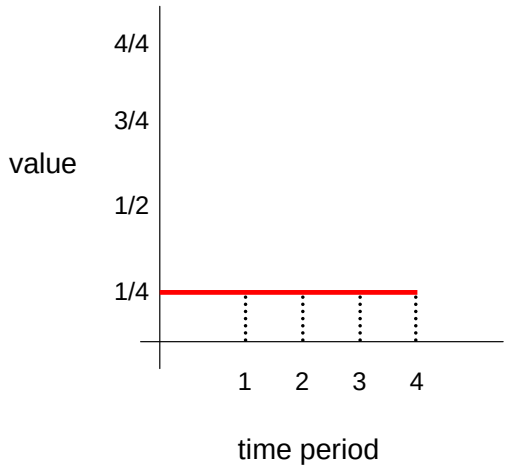


Figure 2: Uniform Utility Curve

Modelers may use a utility function to translate their assumptions about a scenario into a quantity. For example, a utility function that has the shape given in Figure 3 would provide the behavior of all costs being allocated in the first period and zero thereafter. The modeler can select a curve whose shape matches the assumptions about the scenario.

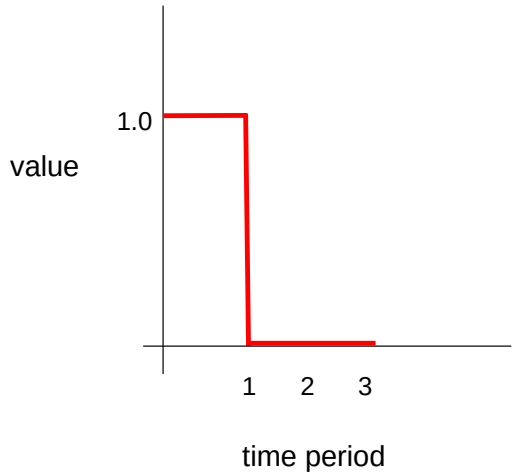


Figure 3: Utility Function

These utility functions can be used to help define the cost functions. For example, C_{org} can be defined by estimating the total cost and then associating with it a utility function such as Figure 4, where two-thirds of the cost are incurred in the first time period and one-third in the second time period.

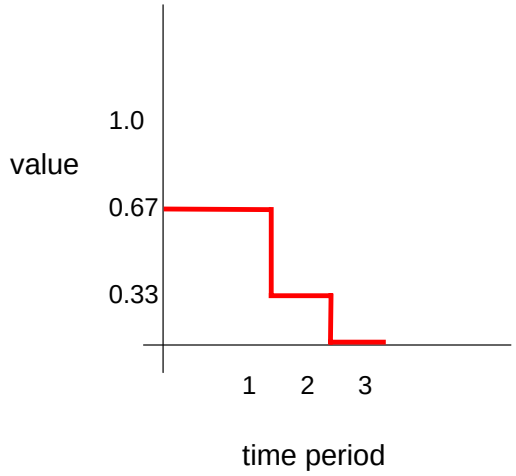


Figure 4: Step Utility Function

Utility functions are most often used as a multiplier of an estimated or predicted cost. In an equation, a modeler can include a term of this form:

cost*u(t)

This term will produce the value of the cost for each iteration. For example, consider a scenario in which the total cost of moving an organization, C_{org}, is $150,000. In this scenario, the modeler selects the utility function shown in Figure 4. The cost term cost*u(t) produces

values of $100,000 in the first period, $50,000 in the second period, and zero for any period thereafter. The modeler changes assumptions by changing the utility function.

4.4 Quantifying Non-Numeric Values

The SEI Cost Benefit Analysis Method (CBAM) is a technique for performing a cost/benefit analysis on architecture decisions [Kazman 02]. It includes an approach to quantifying the benefits of architecture decisions.

4.5 Divide and Conquer

The definitions of the cost functions may be defined by decomposing their definition into other functions. For example, one factor contributing to $C_{cab}()$ is the cost of establishing the product line's software architecture. This cost can, in turn, be decomposed further. Peterson suggests the following list [Peterson 04]:

- **domain analysis:** Domain analysis is concerned with understanding and documenting the requirements commonalities and variabilities.
- **target architecture:** The target architecture specifies the common component boundaries, scope, interfaces, and variation points.
- **technology standards:** A common set of technology standards must be agreed on that deal with operating systems, programming languages, database management systems, graphical user interface technologies, and user interface "look and feel" standards.
- **current architecture:** In order to establish realistic architecture migration plans, an understanding of the current architectures employed across the product family is needed.
- **migration strategy and plan:** The architecture migration strategy and plan make up the roadmap for evolving the existing product family to the target architecture.

Similarly, $C_{org}()$ can be decomposed into the cost of training, the cost of data collection, the cost of reorganization, the cost of establishing necessary processes, and so forth. Of course, the whole cost function approach of SIMPLE is predicated on a "divide and conquer" technique that merely extends the idea.

4.6 Inference

Sometimes it is possible to infer the value of a cost function if you know key information that relates to it. For example, $C_{cab}()$ can be estimated if you know the reuse level throughout a product line—that is, how much of each product on average comes from the core asset base. For example, if the reuse level is 70% (a modest figure for a software product line), the core asset base is going to shoulder the burden, on average, of 70% of each product. If each product costs about the same to build on a stand-alone basis (i.e., has the same $C_{prod}()$ value),

the core asset base is going to shoulder about 70% * $C_{prod}()$ of the cost. But the cost of making software reusable is about 150% of making it in the first place, so C_{cab} will be 150% * 70% * $C_{prod}()$.[11]

Now we have two new functions to calculate: (1) how much it costs to build reusable software (the $F_{reusable}$ function) and (2) how common the products are (the F_{common} function). However, these functions may be easier to project than C_{cab} by itself.

Similarly, C_{reuse} is the cost of reusing reusable artifacts, which according to the literature can be up to 15% of the cost of building them from scratch. Given that the core asset base bears a portion of the burden of any product equal to F_{common}, to get that proportion of a product requires an investment of F_{common} * C_{prod} * 15%. So if F_{common} is 70%, the cost of reusing the core asset base is 15% of 70% of the stand-alone cost of the product.

4.7 Gross Approximation

If first-order approximations reveal an overwhelming preference among alternatives on the table, further modeling may be unnecessary. An example of a rough approximation is assuming that C_{cabu} is some fixed fraction of C_{cab}. This approximation says that the cost of updating the core asset base after each product release is (for example) 10% of the cost of building it—not an unreasonable assumption. Similarly, a modeler might set C_{evo} to some fixed fraction of C_{prod}, saying that evolving a product costs (for example) 15% of building it in the first place.

4.8 Benefit Function Implementation

It is beyond the scope of this report to discuss in detail how the benefit functions are implemented; however, we will provide an example. Consider the benefit of increased customer satisfaction. One technique for quantifying it would follow these steps:

- Develop a classification for customers so that they form homogeneous groups with respect to the dimensions of products for which we are measuring satisfaction (e.g., price or feature sets).
- Collect data about the satisfaction of customers and the likelihood of them purchasing from the company again.
- Maintain data about the average purchase level for each classification.
- After the product line has been implemented, measure the increase (decrease) in customer satisfaction per group. Multiply the increase in satisfaction, as a percentage, by

[11] This approximation is full of assumptions, such as (a) all products are roughly the same in stand-alone cost and (b) the core asset base accounts for the *same* 70% in each product. If those assumptions do not hold, C_{cab} would have to be calculated using product-by-product costs.

the average purchase per group. Sum for all classes to determine the customer satisfaction benefit.

Consider an automobile manufacturer who wishes to add a mass-customizable driver information system. The company monitors three categories of customers: personal use, internal business use, and rental use. The level of satisfaction is recorded in Table 1 as the percentage of customers likely to purchase again.

Table 1: Initial Data

Category	Level of Satisfaction	Average Annual Purchase
Personal use	95%	$25,000
Internal business use	80%	$200,000
Rental use	85%	$2,000,000

After the product line is produced, customer data is collected again (or predicted through marketing surveys). The new data is shown in the third column of Table 2.

Table 2: Satisfaction After the Product Line

Category	Level of Satisfaction	New Level of Satisfaction	Average Annual Purchase
Personal use	95%	95%	$25,000
Internal business use	80%	90%	$200,000
Rental use	85%	90%	$2,000,000

The benefit is calculated in Table 3. The $120,000, in the last line, is the result returned by the benefit function.

Table 3: Benefit Calculation

Category	Level of Satisfaction	New Level of Satisfaction	Increase in Satis-faction	Average Annual Purchase	Expected Increase in Sales Due to Increased Customer Satisfaction
Personal use	95%	95%	0	$25,000	0
Internal business use	80%	90%	.1	$200,000	20,000
Rental use	85%	90%	.05	$2,000,000	100,000
Total					**$120,000**

There are other ways of quantifying customer satisfaction and a wide range of approaches to quantifying other intangible benefits.

4.9 Implementing the Homogeneity Metric

The homogeneity metric provides an indication of the degree to which a product line is homogeneous, taking into account the fact that not every product exhibits the same commonality. The metric varies from 0 to 1, where 0 indicates that the products are all totally unique and 1 indicates that the products are exactly the same. The Goal Question Metric (GQM) analysis used to derive this metric is shown in Appendix A.

The fundamental premise of GQM is that there is a direct relationship between commonality at the feature level and commonality at the core asset level. That is not always the case, but it is sufficient for our purposes. We also normalize the requirements so that each requirement represents the same amount of effort as the others.

The requirements for a product P_i is the union of the product line requirements that apply to P_i and the requirements unique to product P_i:

$$R_{p_i} = R_{pl_{p_i}} \cup R_{u_{p_i}}$$

We define the homogeneity metric as follows:

$$\text{homogeneity} = 1 - \frac{\sum_{i=1}^{number\ of\ products} | R_{u_i} |}{| R_T |}$$

Notation:
PL = product line
U = unique
R = a functional requirement of a product
R_T = the set of all requirements
$| R_T |$ = the total number of different requirements

This metric measures the percentage of the total set of requirements that are unique. If all the requirements are unique, homogeneity is zero. If they are the same, homogeneity is one.

5 Applying SIMPLE in Practice

Experience with applying SIMPLE in an industrial setting is limited to date, but through an early engagement the following process sketch has emerged:

1. **Hold a workshop.** This step gathers people in the organization who have specific questions they wish to answer via SIMPLE. The goal of the workshop is to establish baseline answers to the following questions:
   a. What are the product lines of interest?
   b. What products are currently involved or planned for the future?
   c. What are the scenarios for which we wish to build a model?
   d. Which alternatives are under consideration?
   e. What are the anticipated benefits of each one?
   f. What are the anticipated costs of each one?
   g. What are the relevant time horizons? (e.g., release dates, testing dates)
2. **Formulate the SIMPLE model.** The modelers construct formulas based on the scenario and the alternatives that it embodies, making sure to account for the anticipated costs and benefits of each alternative.
3. **Review and validate the model.** The modelers present the model to the stakeholders to make sure that their concerns are addressed and that the constructed model will, in fact, answer the key questions of interest.
4. **Establish the "shopping list" for data.** The modelers and the stakeholders must work out the organizational data (such as the cost of building past products or the cost of establishing a training program) that will provide input to the formulas.
5. **Populate the model.** The data is gathered and inserted into the formulas, and the results are calculated.
6. **Review the results.** The modelers make a presentation to the stakeholders, and the next steps are planned.

We expect there to be iteration among the steps above, especially 2, 4, and 5. As mentioned earlier, it may turn out that a coarse-grained first-order result reveals a clear preference among the alternatives contained in the scenario. If so, the model's work is done, and no further effort is required. If not, however, the model may have to be reformulated using more finely grained or more sophisticated cost estimates, which, in turn, will result in more precise or detailed data being collected and used to populate the formulas.

We expect to use this process outline in future engagements, where we will refine, elaborate, and evolve the process as appropriate.

6 Relationship to the SEI *Framework for Software Product Line Practice*

The *Framework for Software Product Line Practice*, Version 4.2 [Clements 04] serves a number of purposes by identifying and defining

- foundational concepts underlying software product lines and the essential activities to consider before developing a product line
- practice areas that an organization developing software product lines must master
- needed guidance to an organization about how to move to a product line approach for software

The Framework organizes the product line practices into three categories as shown in Table 4.

Table 4: Product Line Practice Areas

Software Engineering Practice Areas	Technical Management Practice Areas	Organizational Management Practice Areas
Architecture Definition Architecture Evaluation Component Development COTS[12] Utilization Mining Existing Assets Requirements Engineering Software System Integration Testing Understanding Relevant Domains	Configuration Management Data Collection, Metrics, and Tracking Make/Buy/Mine/Commission Analysis Process Definition Scoping Technical Planning Technical Risk Management Tool Support	Building a Business Case Customer Interface Management Developing an Acquisition Strategy Funding Launching and Institutionalizing Market Analysis Operations Organizational Planning Organizational Risk Management Structuring the Organization Technology Forecasting Training

Product line economics, as defined in this report, directly or indirectly supports many of these practice areas. The "Building a Business Case" practice area directly applies the results of the ROI calculation or comparison between product line and stovepipe costs. In other practice areas, the economic models may provide input to the practice area in the form of information needs. The models inform the "Data Collection, Metrics, and Tracking" practice area of data

[12] Commercial off-the-shelf

collection needs, such as historic costs, growth (in size, features, users), or effort distributions.

Even where costs and ROI calculations are not an essential part of the practice, the decision makers should take economic concerns into consideration. Today, these decisions are often made with unsupportable estimates, but the models make it possible to add some degree of rigor to the estimates. For example, a thorough architecture evaluation will require tradeoffs between qualities such as usability and level of security. A question often asked is: How much security can you afford given the potential for loss? In a product line context, the power of the economic models shown in this report could present the costs of building assets at various levels of security, their impact on usability, and the costs of products built using those assets. These costs could be compared with the costs of developing these capabilities through a stovepipe product. The development organization could use this comparison, along with technical issues of the quality attributes, as part of the architecture evaluation.

6.1 Related Software Product Line Practice Areas

The following table provides a list of the practice areas in which SIMPLE models play a role.

Table 5: Relationship of SIMPLE to Product Line Practices

Practice Area	**Role of SIMPLE Models**
Architecture Evaluation	Cost tradeoffs may be one aspect of an architecture evaluation [Kazman 02]. The product line model provides the basic equation from which the evaluator may construct a profile of costs based on architecture choices or options.
Data Collection, Metrics, and Tracking	The economic model relies on good current and historical data for cost estimates and projections. The models provide planners with the basic indicators that they can use to both track current efforts and determine data for collection to build estimates.
Make/Buy/Mine/ Commission Analysis	This analysis relies heavily on cost comparisons, both short and long term. For example, while an organization may find it economically advantageous to build a core asset in house, the long-term costs of sustaining that asset should also be considered. Similar tradeoffs exist in the other aspects of acquiring core assets or even an entire product line. SIMPLE offers models to deal with each of these circumstances and thus can contribute to all aspects of the analysis, especially when comparing the total product line development costs with the costs of commissioning assets or a product, or developing a complete product line.

Table 5: Relationship of SIMPLE to Product Line Practices (cont'd.)

Practice Area	Role of SIMPLE Models
Scoping	Just as scoping provides boundaries on what's in or out of the product line, scoping will also bound the validity of the economic models. A narrowly scoped product line will generally yield greater accuracy in cost projections—all products tend to share the same commonalities. In a broadly scoped product line, individual products in the product line will have less overlap, and previous cost estimates will have less relevance to a new product. SIMPLE may also be used to determine the ROI based on a varied scope of the product line or alternative partitioning of products into one or more product lines.
Technical Planning	Decision making plays a significant role in planning any aspect of the product line—asset development, product development, or long-term sustainment. Cost comparisons provide input to the decision-making process, along with other considerations. SIMPLE can assist the planners in justifying such plan elements as planning the scope of product line assets, choosing asset development approaches, choosing testing approaches, and scheduling the introduction of assets. It also helps planners carry out all aspects of product planning.
Technical Risk Management	At various stages in the maturity of a product line, SIMPLE may either be a source of technical risk or be used to mitigate risk. When first applied to assess product line viability, SIMPLE uses assumptions about costs and projected product line markets. Only results within the bounds of these assumptions are considered reliable. Once a product line is established, the reliability of costs to develop a new product or to extend the product line scope will be grounded in experience and can be a source of risk mitigation—the existence of the product line provides product cost estimates much more reliably than traditional stovepipe estimation techniques.
Tool Support	SIMPLE can influence and justify the selection of product line support tools. The cost of asset reuse (C_{reuse}) can be substantially lowered through the effective choice and use of tools. However, there may be an increase in organizational costs (C_{org}). Generative technologies can also substantially lower C_{reuse}, but at a higher asset development cost (C_{cab}).

Table 5: Relationship of SIMPLE to Product Line Practices (cont'd.)

Practice Area	Role of SIMPLE Models
Building a Business Case	The product line business case relies heavily on cost benefit analyses that use the ROI as a basis for comparison. SIMPLE provides the business case with results from a variety of approaches for assessing and advocating product line approaches. The business case uses historical costs and data as points of comparison with projections for the product line. With SIMPLE, the business case developer can vary the product line scope definition, cost of building assets, cost of using assets, and potential benefits to demonstrate the validity of a product line approach.
Developing an Acquisition Strategy	SIMPLE may be used by an organization to assess and compare alternative acquisition strategies. Acquisition approaches may split asset development and product development activities between the supplier and acquirer or rely entirely on the supplier for all aspects of product line development. In any case, SIMPLE offers models to support cost benefit analyses across a set of strategies, based on either short- or long-term considerations. The acquirer may also use SIMPLE to make comparisons among potential suppliers or require suppliers to use SIMPLE as the basis for their proposals.
Funding	While funding sources and models vary according to organizational culture and the nature of the software product being developed, accurate cost models are essential in both making and justifying funding. At the enterprise level, SIMPLE offers projections across a set of product lines that may encompass the organization's complete set of product lines. In making funding decisions for the individual product line, SIMPLE offers estimates for alternative product line approaches. Finally, funding restrictions may require choices such as which assets to develop or which improvements to fund. SIMPLE also supports models to assist in these decisions.

Table 5: Relationship of SIMPLE to Product Line Practices (cont'd.)

Practice Area	Role of SIMPLE
Launching and Institutionalizing Operations Organizational Planning	While these practice areas actually apply to other practice areas, as appropriate to the needs and capabilities of an organization, SIMPLE models can especially apply here. SIMPLE models can predict the costs of transition where an organization moves to a higher state of product line sophistication, whether starting a product line effort (launching) or trying to expand and/or improve the scope of an ongoing product line effort (institutionalizing). The organizational costs of a product line, C_{org}, encompass these costs as the impact on the entire organization, and asset and product costs (C_{cab}, C_{reuse}, and C_{unique}) encompass actual development costs to fund the transition. Without accurate estimates, the organization cannot adequately plan for the launch or institutionalization.
Market Analysis	While market analysis examines external factors that determine the success of a product in the marketplace, SIMPLE offers models that can compute the ROI from entering that market. A key ingredient of SIMPLE is the calculation of NP (number of products) and "nbr periods." Market analysis plays a key role in both of these calculations. In addition, using different parameters within SIMPLE, an organization can develop cost benefit analyses for a variety of market strategies.
Technology Forecasting	Technology forecasting examines future trends that may affect a product line. Core technologies—as well as the tools, techniques, methods, and processes used to develop the products and bring them to market—all affect costs and may become factors in SIMPLE models. In a rapidly evolving product line, the costs of sustaining the core asset base must take this rapid change into consideration. These costs will be captured in C_{cab} (where new assets are required to accommodate technology evolution) and C_{reuse} (where the modification of assets may be required). Similarly, improvements in development techniques and tools may lower C_{reuse} but may incur organizational costs, C_{org}, as the costs of incorporating these tools into the development environment.

6.2 Patterns for Software Product Line Practice

Patterns for software product line practice [Clements 02b] help organizations appropriately apply the practice areas described in the Framework. SIMPLE models can support many of

these patterns wherever the patterns require decisions based on economic considerations. The patterns represent common context and common problems/solutions, and the use of SIMPLE provides a standard means of approaching the economic product line decision.

Table 6: Relationship of SIMPLE to Patterns for Software Product Line Practice

Pattern	Contribution of SIMPLE
What to Build	SIMPLE provides models to support the "Market Analysis," "Technology Forecasting," "Scoping," and "Building a Business Case" practice areas within the pattern. By applying the models consistently across the practices, the organization will obtain a clear picture of the external market forces and the impact of their decisions. Moreover, the results are not just a static cost savings or ROI, but can include the dynamics of market and technology impacts over the life of the product line.
Product Parts	This pattern relates decisions of the "Make/Buy/Mine/ Commission Analysis" practice area to other practices and patterns that are used to build or acquire assets. SIMPLE provides the basics for decisions in the "Make/Buy/Mine/ Commission Analysis" practice area and can also establish data needs as produced in other patterns and practices.
Monitor	While SIMPLE is most often used to develop cost benefit analyses or business cases in advance of product line development, the models should also be used in tracking product line performance. Many of the estimates used as parameters in SIMPLE calculations may, during the course of product line development, become hard numbers as actual results are collected and tracked. In the Monitor Pattern, the "listen group" in the "Data Collection, Metrics and Tracking" and "Technical Risk Management" practice areas can provide hard data to the "response group" to improve the accuracy of SIMPLE predictions. Such improvements can be part of revised and new production plans.

Table 6: Relationship of SIMPLE to Patterns for Software Product Line Practice (cont'd.)

Pattern	Contribution of SIMPLE
Cold Start In Motion	In both of these patterns, the "Funding" practice area plays a significant role. Funding sources require justification and validation of the product line approaches, whether initiating a product line or continuing to support it. SIMPLE models provide input to these decision makers by providing accurate pictures of both the current and projected product lines. These pictures are used in the pattern to support choices such as: whether to expand the scope of the product line; whether to split the product line or combine product lines; which assets to invest in; and, in some cases, whether to terminate the product line.

7 Related Work

SIMPLE models the effects of systematic reuse also covered by other techniques, some of which we're already mentioned.

One model of note was proposed by Dale Peterson based on the experience of his company, Convergys, in opting to take a software product line approach [Peterson 04]. Based on an understanding of the requirements' commonalities and variabilities, as well as on the component architecture, Peterson's model attempts to quantify such things as

- how much of the development work will be performed by the component groups versus the product groups
- the benefits associated with improved productivity and the use of common components, based on the concept of overlapping market needs
- the manpower savings associated with eliminating redundant software development and increasing productivity
- the benefits associated with increasing the throughput of the software factory (i.e., by doing more work with the same number of people)

Two other modeling approaches are discussed in detail below: Poulin's *Measuring Software Reuse* [Poulin 97a] and COPLIMO [Boehm 04]. Poulin considers the use of assets in developing individual products and the potential for cost savings, but his models do not take the broader implications of product lines into consideration. COPLIMO is based on the widely used COCOMO II [Boehm 00] and provides an algorithmic approach to cost calculations for product lines.

7.1 Poulin's *Measuring Software Reuse*

Poulin uses two parameters in estimating the effects of a systematic reuse strategy: (1) the relative cost of reuse (RCR) and (2) the relative cost of writing for reuse (RCWR) [Poulin 97a, Poulin 97b]. The RCR is a ratio that compares the effort needed to reuse software without modification to the costs normally associated with developing that same software for a single use. An RCR of 0.1 for an asset, or set of assets, says that the cost of using that asset is one-tenth the cost of developing equivalent software for a single use. The costs of creating that asset are reflected in the RCWR. This value relates the costs of creating reusable software to the cost of writing one-time-use software. For example, an RCWR of 1.5 represents the addition of 50% to the effort of creating single-use software—if it costs $10,000 to create a component for a single use, a reusable version would cost $15,000, accounting for increased design time, testing, documentation, and so forth.

The Poulin model uses the RCR and RCWR to calculate two more values that predict savings over an entire project:

1. The reuse cost avoidance (RCA) compares the savings of reusing assets over writing the equivalent software for a single use. The RCA incorporates two values: the development cost avoidance (DCA) and the service cost avoidance (SCA). To calculate these values, Poulin uses the reused source instructions (RSI). The RSI value comes from the level of reuse on a project where

$$\%reuse = \frac{reused\_source\_instructions}{total\_source\_instructions} * 100\%$$

   Given the RSI, Poulin calculates the DCA by calculating the cost avoidance based on the total software reused and the cost of reusing that software:

   DCA = RSI * (1-RCR) * (Cost of single-use code/lines of code [LOC])

   For example, if the RCR is 0.1, and the reused software is 10 thousand lines of code (KLOC), and the typical costs per LOC is $40

   DCA = 10000 * (1-0.1) * ($40) = $360,000

   The SCA represents the maintenance savings accrued through the elimination of repair costs:

   SCA = RSI * (Error rate) * (Error cost)

   For example, if error rates are 1.5 errors/KLOC and error costs are $1000/error

   SCA = 10000 LOC * 1.5 errors/KLOC * $1000/error

   = 15 errors * $1000/error

   = $15,000

   The RCA in this example is DCA + SCA, or $375,000.

2. The additional development cost (ADC) is the cost of writing software for reuse and is based on the RCWR and the actual code written for reuse. The ADC reflects the cost of writing the reusable software, reflected in the RSI, over the cost of writing for a single use and is expressed as

   ADC = (RCWR -1) * RSI * (new code cost)

   For example, using the numbers above

   ADC = (1.5 – 1) * 10 KLOC * ($40/LOC) = $200,000

Under this approach, Poulin calculates the ROI as

$$ROI = \sum_{i-1}^{n} RCA_i - ADC$$

where *i* represents the successive uses of the reusable software. Assuming a series of 3 development efforts using these 10 KLOC for reuse

$$\text{ROI} = (\$375{,}000 * 3) - \$200{,}000 = \$925{,}000$$

Poulin looks only at the cost savings based on estimated development and reuse costs. Because Poulin's method involves a reuse model and not a product line model, it can be used in a very limited fashion compared to SIMPLE. Moreover, Poulin's method does not look at the effects of time on the reuse of assets and necessary changes to those assets or consider alternative strategies for developing and using a product line.

7.2 COPLIMO

COPLIMO is an outgrowth of COCOMO. Both models estimate effort based on code size plus a combination of development factors. For COCOMO, size represents KLOC. For COPLIMO, the model applies a series of equations to generate an "equivalent code size,"—that is, given asset use in a product line, there is some value that represents a code size for a product. This product line code size takes into consideration the costs of generating the assets, the complexity of the assets, the effort of using the assets, and other factors.

COCOMO produces a nominal effort for product development that is defined as

$$\text{nominal effort} = a * (\text{size})^b$$

In this equation, the factors a and b are dependent on the relative complexity of a software product—a more complex product requires additional effort, all other factors being equal.

A final effort calculation in COCOMO requires the inclusion of a set of effort multipliers—that is, factors that influence cost. Examples of these multipliers include the experience of the development organization, level of documentation, use of tools, and other development factors. Given these multipliers, the development effort in COCOMO is defined as

$$\text{effort} = \text{nominal effort} * \prod_{i=1}^{n} effort\_multiplier_i$$

COPLIMO uses a similar equation but must develop two components in order to apply the equation:

- a cost model for product line development
- an annualized post-development life-cycle extension

7.2.1 Product Line Development in COPLIMO

The cost model for product line development calculates values similar to the RCWR and RCR from Poulin. For the RCWR, COPLIMO uses multipliers from COCOMO, namely

development for reuse (RUSE) and two constraints from COCOMO: (1) required reliability (RELY) and (2) degree of documentation (DOCU). These factors represent the added costs of developing product line assets. COPLIMO uses 15-20% as the added effort of developing assets for reuse (an effort that is lower than Poulin because COCOMO removes other contributing factors) and then applies the other constraints:

estimated cost = (1 + RUSE) * cost of reliability * cost of documentation

COPLIMO calculates the RCR as an equivalent size of code using a factor known as assessment and assimilation (AA). The RCR calculations consider two cases:

1. Black-box reuse applies the AA factor in the range 0.02 < AA factor < 0.08, based on the effort of reusing the code. When multiplied by the amount of code reused, this factor generates an equivalent size. For example, if there is no effort to reuse the code, except to find it, the total code is multiplied by 0.2. Where testing and evaluation are required, the factor may be as high as 0.08. For 10 KLOC reused, the equivalent size could range from 200 to 800, depending on the AA factor.
2. Adapted software (white-box reuse) includes several factors:
   - the amount of design modified (DM) and code modified (CM)
   - the integration effort (IM)
   - the system understanding factor (SU) and programmer unfamiliarity (UNFM)

The formula first accounts for modification (adaptation adjustment factor [AAF]) as the design and code modification plus integration effort:

- AAF = (.4*DM) + (.3*CM) + (.3*IM)
- COPLIMO calculates an adaptation adjustment multiplier (AAM). Where AAF is <= 50, AAM =

$$\frac{AA + AAF(1 + (0.02 * SU * UNFM))}{100}$$

Where AAF > 50, AAM =

$$\frac{AA + AAF + (SU * UNFM)}{100}$$

In these formulae, system understanding will be lower based on the clarity of the architecture, a clear match between assets and product, and the level of documentation; unfamiliarity will be lower based on the frequency of asset use.

The AAM is then multiplied by the LOC reused to obtain an equivalent size of code for further COPLIMO calculation. For moderate (25%) levels of modification and high levels of system understanding and unfamiliarity, the AAM will be approximately 30%. Modifying 10 KLOC under these conditions would be the equivalent of creating 3 KLOC from scratch.

COPLIMO then takes equivalent sizes and applies standard COCOMO multipliers. If the product consists of 30 KLOC, 10K of which are new, 10K are reused as is, and 10K are reused with modifications, the equivalent lines of code would equal

- 10,000 (for the new lines) +
- 10,000 * 8/100 = 800 lines of black-box reuse (where 8/100 represents the cost AAF)
- 10,000 * .3 = 3000 lines of modified reuse (where .3 is the AAM as shown above)

The equivalent lines would total 13,800. Given this value, COCOMO can be applied as stated above:

$$\text{effort} = A * (\text{PSIZE})^{B} * \prod_{i=1}^{n} effort\_multiplier_i$$

You can compare the cost of development from scratch with PSIZE = 30 (COCOMO uses KLOC for size) versus 13.8 for the reuse-based approach. Using typical COCOMO values for A (e.g., 2.94) and 1.0 for B and all effort multipliers, this formula give values of approximately 90 person-months (PM) and 40PM respectively for from-scratch versus product line development. The cost savings must take product line development costs into account, as well, which will cover the 20 KLOC to be reused. Given typical values (e.g., 1.2) for RUSE, DOCU, and RELY, the asset development costs could be 1.2*1.2*1.2 = 1.7 times those of development from scratch. COCOMO would predict approximately 100PM of effort to develop the assets. After three similar applications of the asset base, the COPLIMO model would show approximately 50PM return, 270 (90 * 3)PM to develop three products from scratch, versus 220PM (100 + 40 + 40 + 40) using the product line assets.

7.2.2 Annualized Life-Cycle Model in COPLIMO

COMPLIO makes some simplifying assumptions when looking at the life cycle of maintaining the core asset base and products. Three factors must remain relatively constant:

1. the fractions of a product that are new, reused, or adopted
2. all the effort multipliers (e.g., AA, RUSE, AAM)
3. the annual change traffic (ACT) or the fraction of a product's software that changes per year

In calculating life-cycle costs, COPLIMO uses the basic COCOMO approach: add the initial development cost (in PM) to the maintenance costs based on the number of years in maintenance. The maintenance costs use an annual maintenance size based on product size, the ACT, and system understanding and unfamiliarity factors. The actual formula for size is

$$\text{AMSIZE} = \text{PSIZE} * \text{ACT}\left(1 + \frac{SU}{100} * \text{UNFM}\right)$$

COCOMO factors the AMSIZE into a maintenance formula as

$$PM(N, L) = PM(N) + L * N [A * (AMSIZE)^{B} * \prod (EM)]$$

In this formula, N is the number of products under maintenance and L is the number of years. For the product line, COPLIMO applies this same formula once for each of the three categories of new, reused, and adopted assets. The results are summed to give a total maintenance value.

As with Poulin, COPLIMO is essentially a reuse-based model: COPLIMO assumes the use of a set of assets for building a set of related products. COPLIMO goes further than Poulin by considering variations in the cost of reuse (C_{reuse} in SIMPLE) and in considering maintenance but makes some simplifying assumptions as well. On the other hand, COPLIMO relies on the availability of a range of parametric values that must be accurately calibrated, making it less suitable for a nontechnical audience.

8 Future Work

SIMPLE is still very much a work in progress, with four main efforts remaining:

1. validating the model
2. adding new scenarios and sorting out how to organize them
3. exploring dependencies and sensitivities within the model
4. making SIMPLE widely available and usable

Each effort is described below.

8.1 Validating SIMPLE

How will we know that the predictions returned by SIMPLE are correct? The fact is that most likely, they will not be. All models are subject to accuracy and precision concerns, especially SIMPLE. So the appropriate question becomes, "How will we know if the answers returned by SIMPLE are good enough?"

What does it mean for a prediction to be good enough? If a model predicts 42 and the real answer is 142, is the model good enough or not? It is a question that can be answered only in the context of goals for the model. If a model that is accurate within an order of magnitude is a good one (i.e., if it returns helpful results), the model that returns 42 is indeed a good one. But if only answers within 5% of reality are helpful, the model is not a good one.

We envision that SIMPLE will be used by product line champions as a tool to provide first-order-of-magnitude feasibility results related to envisioned product line scenarios. As one reviewer put it, "If an organization can realize a productivity gain of 200-500% through a product line approach, you don't need to worry if your model is off by some small integer factor. You aren't trying to convince someone who's worried about whether some value is 12% or 13%."

With this in mind, our task becomes one of gaining confidence in the model, not validating it in the strict sense. Towards this end, we are making arrangements with one organization to cooperatively formulate several scenarios of interest, make SIMPLE predictions, and then compare those predictions to actual results over time. We are also looking for other organizations with which to repeat this process. As we do this, we will publish the results so that potential users of SIMPLE can see how well those pilot organizations fared and decide on their own whether the results warrant its use.

8.2 Expanding and Organizing the Set of SIMPLE Scenarios

This report has presented a number of scenarios that SIMPLE can calculate. Of course, many more are possible. We'd like to try additional scenarios that

- explore the short-term benefits but long-term costs of the "clone and own" approach to exploiting commonality
- express time-based options, such as choosing between building core assets heavily early on versus spreading out the process over time. This kind of scenario would address *incremental* product line development [Clements 04].
- express choices between different production strategies, such as using a generator to produce products versus using more traditional check-out/parameterize/compile/build approaches. This kind of scenario would let an organization investigate the feasibility of a generative programming approach to its product line [Czarnecki 00].
- let users investigate the difference between reactive and proactive approaches to product line engineering [Clements 02c]
- let users investigate different approaches to providing variability in their core assets. One way of doing that is to ask, "How well is an envisioned product supported by the variability provided?"
- investigate different strategies for product line testing
- let users choose which of hundreds of change requests should be given highest priority

We would also like to provide a systematic means for organizing scenarios so that SIMPLE users can find the scenarios they want easily without having to search randomly through a long unstructured list.

8.3 Exploring Intra-Model Dependencies and Sensitivities

Dependencies certainly exist among the variables that constitute a SIMPLE model for a product line. For instance, providing a core asset base whose assets come equipped with wizards to guide instantiation and installation will increase C_{cab} but reduce C_{reuse}. And if C_{cab} provides core assets that cover most of a product, C_{unique} will be low, but C_{cab} may well be somewhat high. Uncovering and quantifying these dependencies will help a model builder set up the model, check it for reasonableness, and perform various "what if" investigations.

Also, it may be the case that a model for a particular product line addressing a particular scenario is much more sensitive to the value of some variables than others. Knowing that will tell a modeler which variables are important to have accurate estimates for and which variables can be filled in with more "finger in the wind" guesses. Peterson describes a good example of analyzing a model for its input sensitivities [Peterson 04].

8.4 Presentation Issues: Making SIMPLE Available and Usable

We envision a Web-based user interface for SIMPLE. Product line decision makers who wish to use the model will visit a Web site where they will be presented with a number of scenarios for which SIMPLE formulas have been developed. After choosing the one that best describes their situation of interest, they will be asked to fill in estimated values of the parameters relevant to that scenario. The formulas will be run, and answers will be provided.

Longer term plans might include

- displaying a number of graphs, each showing how one of the outputs is a function of one or more of the input parameters
- accepting a range of inputs, as a hedge against uncertainty, with the result of having SIMPLE compute a range of outputs
- using existing skeletal outlines for a business case [Cohen 01] to produce prepackaged business case write-ups on request that incorporate the results of choosing the scenario and running the model
- preparing a list of assumptions (some inherent in the model, some inherent in the selected scenario, and some inherent in the inputs provided) that the results encompass
- addressing concerns about submitting confidential data to a Web site by, for example, providing ways to let users download the computational software
- letting users suspend and resume a modeling session over time
- data farming, or collecting (without retaining any identification of the user) entered profiles, so as to gain insights into the kinds of product line scenarios and values that are being experienced in the real world
- showing role-based benefits

Still longer term plans might include a section of the Web site where visitors can propose new scenarios and then propose SIMPLE formulations that express those scenarios. Much like open source software, these scenarios and their formulas would be built and maintained by the user community, who could decide whether the formulations were reliable enough to use.

Appendix A – Goal Question Metric (GQM) Analysis of Homogeneity Metric

Several activities in product line management depend on the characteristics of the products in the product line. The degree of reuse among the products in the product line depends on how homogeneous the products are. In this report, we propose a metric that characterizes the homogeneity of a set of products. This metric can be used for various estimation and planning activities such as product line scoping.

We use the GQM approach [Basili 92] to describe the context and basic definition of the metric. We then give a complete definition of it and illustrate its use with several scenarios.

Goal

Our goal is to be able to characterize a product line, so we can compare one product line to another. A product line made up of wireless devices that differ from one another only by the types of peripheral devices that attach to it is different from a product line made up of handheld terminals that perform diverse tasks such as traffic ticket issuance or merchandise inventory control. A higher level of component reuse would be anticipated in the former product line than in the latter.

Questions

The statement of the goal raises several questions:

1. How many products are in each product line?
2. What is the size of each product in the product line?
3. How complex is each product and, by inference, the product line?
4. How similar are the products within a single product line? The more similar the products, the higher the degree of reuse and the faster the time to market for the product line.

Metric

To answer the question posed above about the similarity of the products in a product line, we need to define a measure that characterizes how homogeneous the products are. To do that, we must address the issues discussed below.

Units

The first issue is the selection of the unit that will be used in the computation of the metric. A product is most often thought of in terms of its requirements; however, different writers will divide the responsibilities of a system differently, resulting in potentially different values of the metric for the same product. Two requirements may have very different impacts on the system, yet a counting scheme may count each as a single requirement. There is no uniform technique for writing requirements.

Two other candidate units are features and use cases. A feature is just as nebulous as a requirement, but techniques such as feature-oriented domain analysis (FODA) [Kang 90] can reduce the variation between people defining the same system. Likewise, use cases can be vague, but specific techniques [Major 98, Cockburn 01] exist that result in more uniform use cases.

Following the approach used by McGregor [McGregor 95], the conceptual definition of the metric can remain unchanged, while different units of measure are selected. This approach allows measures of homogeneity to be defined very early in the product line life cycle and on into the implementation of the products. So early on, features or use cases can be considered the units, while later, unique components can be the units.

Accuracy

In defining the measure, a number of issues will arise that affect the accuracy of the measure. For example, are all the requirements, features, or other units equally important? What if one requirement is a derivative of another?

We are taking an iterative approach. Initial results from using the metric will drive modifications in the definition.

Metric Definition

Consider the definition of the following measure:

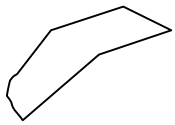
Notation:
NP = number of products in the product line
R = a functional requirement of a product

R_T = the set of all requirements
$| R_T |$ = the total number of different requirements

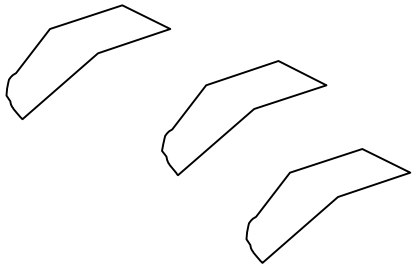
Assumption: A "unique" requirement is one that is used by only one product.

Scenario: Suppose there are 10 products in the product line and a total of 100 separate requirements. There are several possibilities:

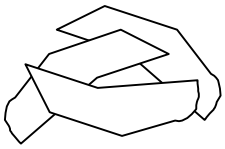
a) All requirements apply to all products.



b) Ten requirements apply to each product, and each product is different.



c) Some products share requirements, but not the same requirements.



The requirements for a product P_i is the union of the product line requirements that apply to P_i and the requirements unique to product P_i.

$$R_{p_i} = R_{pl_{p_i}} \cup R_{u_{p_i}}$$

So we need a measure for the ***homogeneity*** of the product line. Consider the measure

$$\text{homogeneity} = 1 - \frac{\sum_{i=1}^{number\ of\ products} | R_{u_i} |}{| R_T |}$$

There are three cases to consider:

1. In the degenerate case where every product is totally unique

   $| R_{pl} | = 0$ and

   $| R_T | = \sum_{i=1}^{number\ of\ products} | R_{u_{p_i}} |$ then

   $$\text{homogeneity} = 1 - \frac{| R_T |}{| R_T |} = 0$$

   When each product is unique, the number of products using any requirement is 1. Each term in the sum is $\frac{1}{NP}$, the sum is $\frac{T}{NP}$, and the result is 1 - $\frac{1}{NP}$.

2. If all the products are identical, $| R_T | = | R_{pl} |$ and $\sum_{i=1}^{number\ of\ products} | R_{u_{p_i}} | = 0$ and

   homogeneity = 1 – 0 = 1.

   When the products are identical, each term in the numerator is a 1. The sum is T, the quotient is 1, and homogeneity is 1 – 1 = 0.

3. This still leaves the third case in which a requirement applies to more than one product but not all of them. This is still a product line requirement, but the variables of diversity and reuse factor still can vary over wide ranges. We consider an average case where all requirements are used by more than one product, but none are used by all products.

   One possibility is to weight each requirement by the fraction of the total number of products that use it.

   $$\text{homogeneity} = \frac{\sum_{i=1}^{T} \frac{number\ of\ \ products\ satisfying\ R_i}{NP}}{| R_T |}$$

Consider these scenarios:

- Suppose there are 100 requirements and 10 products and a core set of 10 requirements common to all products. Each product has nine unique requirements. In this case, the sum is 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 90(.1) = 19, and the homogeneity is 1 - 19/100 = .81

- Suppose there are 100 requirements and 10 products and a core of 50 requirements common to all products. Each product has five unique requirements. In this case, the sum is 50(1) + 50(.1) = 55, and the homogeneity is 1 - 55/100 or .45.
- Suppose there are 100 requirements and 10 products and a core of 80 requirements common to all products. Each product has two unique requirements. In this case, the sum is 80(1) + 20(.1) = 82, and the homogeneity is 1 - 82/100 or .18.

Appendix B – Table of Symbols Used in SIMPLE

Symbol	Definition
B_{ben_j}	A benefit function that returns the economic value of product line benefit ben_j (such as faster time to market)
C_{cab}()	A cost function that, given the relevant parameters, returns the cost of developing a core asset base suited to satisfy a particular scope. C_{cab} takes into account the costs of performing a commonality/variability analysis, defining the product line's scope, designing and then evaluating a generic (as opposed to one-off) software architecture, and developing the software so designed. C_{cab} may be invoked to tell us the cost of developing a core asset base where none currently exists or deriving a desired core asset base from one or more already in place.
C_{cabu}()	The cost of updating the core asset base has to change as a result of releasing a new version of a product
C_{evo}()	A cost function that is parameterized with product and version numbers and returns the cost of producing that version without using a core asset base
C_{org}()	A cost function that, given the relevant parameters, returns how much it costs an organization to adopt the product line approach for its products. Such costs can include reorganization, process improvement, training, and whatever other organizational remedies are necessary.
C_{prod} (*product*)	A cost function that returns the cost of building a product in a stand-alone fashion
C_{reuse}()	A cost function that, given the relevant parameters, returns the development cost of reusing core assets from a core asset base to build a product. C_{reuse} includes the cost of locating a core asset, checking it out of the repository, tailoring it for use in the intended application, and performing the extra tests associated with reusing core assets.
C_{unique}()	A cost function that, given the relevant parameters, returns the cost of developing the unique parts (both software and non-software) of a product that are not based on the assets in the core asset base. The result might be a complete product (i.e., one that is not a member of a product line) or the unique part of a product whose remainder is built atop the core asset base of a product line.
F_{common}	A factor describing what fraction of a product (or, on average, a set of products) is built from core assets (e.g., 70%)
$F_{reusable}$	A factor describing how much more it costs to build multi-use software than single-purpose software (e.g., 150%)
ROI	return on investment, calculated as *cost savings/cost of investment*
t	A time period of interest, used as a parameter to cost and benefit functions

Appendix C – Acronym List

Acronym	Definition
AA	assessment and assimilation
AAF	adaptation adjustment factor
AAM	adaptation adjustment multiplier
ACT	annual change traffic
ADC	additional development cost
CBAM	Cost Benefit Analysis Method
CM	code modified
COCOMO	Cost Construction Model
COPLIMO	Constructive Product Line Investment Model
COTS	commercial off-the-shelf
DCA	development cost avoidance
DM	design modified
DOCU	degree of documentation
GQM	Goal Question Metric
FODA	feature-oriented domain analysis
IM	integration effort
KLOC	thousand lines of code
LOC	lines of code
PM	person-months
PY	person-years
R&D	research and development
ROI	return on investment
RCA	reuse cost avoidance
RCR	relative cost of reuse
RCWR	relative cost of writing for reuse
RELY	required reliability
RSI	reused source instructions
RUSE	development for reuse

Acronym	Definition
SCA	service cost avoidance
SEI	Software Engineering Institute
SIMPLE	Structured Intuitive Model of Product Line Economics
SU	system understanding
UNFM	programmer unfamiliarity

References

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

[Basili 92] Basili, Victor R. *Software Modeling and Measurement: The Goal/Question/Metric Paradigm* (CS-TR-2956, UMIACS-TR-92-96). College Park, MD: University of Maryland, September 1992.

[Böckle 04a] Böckle, Günter; Clements, Paul; McGregor, John D.; Muthig, Dirk; & Schmid, Klaus. "Calculating ROI for Software Product Lines." *IEEE Software 21*, 3 (May/June 2004): 23-31.

[Böckle 04b] Böckle, Günter; Clements, Paul; McGregor, John D.; Muthig, Dirk; & Schmid, Klaus. "A Cost Model for Software Product Lines," 310-316. *Proceedings of the 5th International Workshop on Product Family Engineering (PFE 2003)*, Lecture Notes in Computer Science (LNCS) 3014. Siena, Italy, November, 2003. Berlin, Germany: Springer, 2004.

[Boehm 04] Boehm, Barry; Brown, A. Winsor; Madachy, Ray; & Ye Yang. "A Software Product Line Life Cycle Cost Estimation Model," 156-164. *Proceedings of the 2004 International Symposium on Empirical Software Engineering*. Redondo Beach, CA, August 19-20, 2004. Los Alamitos, CA: IEEE Computer Society, 2004.

[Boehm 00] Boehm, Barry W. *Software Cost Estimation with Cocomo II*. Upper Saddle River, NJ: Prentice Hall, 2000.

[Clements 04] Clements, Paul & Northrop, Linda. *A Framework for Software Product Line Practice, Version 4.2*. http://www.sei.cmu.edu/productlines/framework.html (2004)

[Clements 02a] Clements, Paul & Northrop, Linda. *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*. Boston, MA: Addison-Wesley, 2002.

[Clements 02c] Clements, Paul & Krueger, Charles. "Initiating Software Product Lines - Point/Counterpoint: Being Proactive Pays Off." *IEEE Software 19*, 4 (July/August 2002): 28.

[Cockburn 01] Cockburn, Alistair. *Writing Effective Use Cases*. Boston, MA: Addison-Wesley, 2001.

[Cohen 04] Cohen, Sholom; Zubrow, Dave; & Dunn, Ed. *Case Study: A Measurement Program for Product Lines* (CMU/SEI-2004-TN-023). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2004. http://www.sei.cmu.edu/publications/documents/04.reports/04tn023.html

[Cohen 03] Cohen, Sholom. *Predicting When Product Line Investment Pays* (CMU/SEI-2003-TN-017, ADA418466). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. http://www.sei.cmu.edu/publications/documents/03.reports/03tn017.html

[Cohen 01] Cohen, Sholom. *Case Study: Building and Communicating a Business Case for a DoD Product Line* (CMU/SEI- 2001-TN-020, ADA395155). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001. http://www.sei.cmu.edu/publications/documents/01.reports/01tn020.html

[Cruickshank 93] Cruickshank, R. D. & Gaffney, J. E. "An Economic Model of Software Reuse," 99-137. *Analytical Methods in Software Engineering Economics*. Edited by T. Gulledge & W. Hutzler. New York, NY: Springer-Verlag, 1993.

[Czarnecki 00] Czarnecki, K. & Eisenecker, U. *Generative Programming: Methods, Tools, and Applications*. Boston, MA: Addison-Wesley, 2000.

[Favaro 96] Favaro, John. "A Comparison of Approaches to Reuse Investment Analysis," 136-145. *Proceedings of the Fourth IEEE International Conference on Software Reuse*. Orlando, FL, April 23-26, 1996. Los Alamitos, CA: IEEE Computer Society Press, 1996.

[Gaffney 92] Gaffney, J. E., Jr. & Cruickshank, R. D. "A General Economic Model of Software Reuse," 327-337. *Proceedings of the International Conference on Software Engineering*. Melbourne, Australia, May 11-15, 1992. New York, NY: ACM, 1992.

[ISO9126 01] International Organization for Standardization (ISO) & International Electrotechnical Commission (IEC). *Software Engineering—Product Quality = Genie du Logiciel—Qualite des Produits* [ISO/IEC 9126-1:2001(E)]. Geneva, Switzerland: ISO/IEC, 2001.

[Kang 90] Kang, K.; Cohen, S.; Hess, J.; Novak, W.; & Peterson, A. *Feature-Oriented Domain Analysis (FODA) Feasibility Study* (CMU/SEI-90-TR-021, ADA235785). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990. http://www.sei.cmu.edu/publications/documents/90.reports/90.tr.021.html

[Kazman 02] Kazman, Rick; Asundi, Jai; & Klein, Mark. *Making Architecture Design Decisions: An Economic Approach* (CMU/SEI-2002-TR-035, ADA408740). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2002. http://www.sei.cmu.edu/publications/documents/02.reports/02tr035.html

[Knauber 02] Knauber, Peter; Bermejo, Jesus; Böckle, Günter; Leite, Julio; van der Linden, Frank; Northrop, Linda; Stark, Michael; & Weiss, David. "Quantifying Product Line Benefits," 155-163. *Proceedings of the 4th International Software Product Family Engineering Workshop (PFE 2001)*, Lecture Notes in Computer Science (LNCS) 2290. Bilbao, Spain, October 3-5, 2001. Berlin, Germany: Springer-Verlag, 2002.

[Major 98] Major, Melissa & McGregor, John D. *A Qualitative Analysis of Two Requirements Capturing Techniques for Estimating the Size of Object-Oriented Software Projects* (Department of Computer Science Technical Report 98-002). Clemson, SC: Clemson University, 1998.

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

[McGregor 95] McGregor, John D. "Managing Metrics in an Iterative Environment." *Object Magazine 5*, 6 (1995): 65-71.

[Mili 00] Mili, A.; Chmiel, S. F.; Gottumukkala, R.; & Zhang. L. "An Integrated Cost Model for Software Reuse," 157-166. *Proceedings of the 2000 International Conference on Software Engineering*. Limerick, Ireland, June 4-11, 2000. New York, NY: ACM, 2000.

[Peterson 04] Peterson, Dale. "Economics of Software Product Lines," 381-402. *Proceedings of the 5th International Workshop on Product Family Engineering (PFE 2003).* Siena, Italy, November 2003. Berlin, Germany: Springer, 2004.

[Poulin 97a] Poulin, Jeffrey S. *Measuring Software Reuse*. Reading, MA: Addison-Wesley, 1997.

[Poulin 97b] Poulin, Jeffrey S. "Fueling Software Reuse with Metrics." *Object Magazine 7*, 7 (September 1997): 42-47.

[Schmid 02] Schmid, Klaus. "An Initial Model of Product Line Economics," 38-50. *Proceedings of the 4th International Software Product Family Engineering Workshop (PFE 2001)*, Lecture Notes in Computer Science (LNCS) 2290. Bilbao, Spain, October 3-5, 2001. Berlin, Germany: Springer-Verlag, 2002.

[SPL 05a] *Success Stories in Software Product Lines*. http://www.softwareproductlines.com/successes/successes.html (URL valid as of March 2005)

[SPL 05b] *Binding Times*. http://www.softwareproductlines.com /introduction/binding.html (URL valid as of March 2005)

[Verhoef 04] Verhoef, C. *Quantifying the Value of IT-Investments*. http:www.cs.vu.nl/~x/val/val.pdf (August 2004)

[Weiss 99] Weiss, David & Lai, Chi Tau Robert. *Software Product-Line Engineering*. Reading, MA: Addison-Wesley, 1999.

[Wiles 02] Wiles, Ed. “Economic Models of Software Reuse: A Comparison and Validation.” PhD diss., University of Wales. 2002.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

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

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
The Structured Intuitive Model for Product Line Economics (SIMPLE)	F19628-00-C-0003

6. AUTHOR(S)

Paul C. Clements, John D. McGregor, Sholom G. Cohen

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-003

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-2005-003

11. SUPPLEMENTARY NOTES

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

13. ABSTRACT (MAXIMUM 200 WORDS)

Software product line practice is an effective strategy for developing families of software-intensive products. Business modeling is a fundamental practice that provides input into a number of decisions that are made by organizations using or considering using the product line strategy. This report presents the Structured Intuitive Model of Product Line Economics (SIMPLE), a general-purpose business model that supports the estimation of the costs and benefits in a product line development organization. The model supports decisions such as whether to use a product line strategy in a specific situation, the specific strategy to apply, and the appropriateness of acquiring or building specific assets. This report illustrates the model's scope by presenting several scenarios and its usefulness by integrating it into several product line practice patterns. The report ends with a description of future work aimed at making the model usable by product line practitioners.

14. SUBJECT TERMS	15. NUMBER OF PAGES
software product line, cost model, return on investment, ROI, product line economics, product line business case	80

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