SHARE VS. OWN
Software Reuse using Product Platforms

Paul Abraham, Vishal Sikka
Office of the CTO, SAP AG., Dietmar Hopp Allee, NeurottStrasse16, D-69190 Walldorf, Germany

Gordon Simpson
Independent Enterprise Technologist, San Francisco, CA, U.S.A.

Keywords: Software reuse, Product platform, Application platform, Technology platform, Product line architecture. Cost-benefit analysis.

Abstract: SAP is a complex multi-national development organization with a large number of diverse products and changing target markets. Effective allocation of resources is a difficult at the best of times. Of late, the target markets, and supporting technologies, change every couple of years exponentially increasing the complexity, necessitating a way of recalibrating that keeps pace with new realties. SAP, with a mature understanding of functional, software and technical relationships, has adopted a platform approach covering both functional and technology capabilities. However, a variety of factors, many in the management space, prevent that from being effective. This paper will explain why product-line/platform is a better strategy than platform or custom product strategies, in a way that can be understood, proven and adopted by management and developers alike. Specific recommendations of practices for delivering reuse effectively are also provided.

1 PROBLEM CONTEXT

SAP spends a great deal of time examining how to allocate resources among its development groups. This allocation is made most efficient by sharing resources among products; dedicating resources only when dictated by necessity. The advent of groundbreaking technology changes or changes in the marketplace force us to re-evaluate this allocation. The acceleration of these previously slow moving factors requires SAP to develop a systematic way of matching pace with evaluation. These change factors and associated resource issues are, to some degree, applicable in turn to SAP‘s ecosystem i.e. development partners, technology vendors and customers.

When, how and what to share, of software components, processes, infrastructure, people and knowledge, will need to be continuously evaluated as SAP products go through several cycles of renovation. SAP has thus far adopted reuse primarily through a product platform strategy, using both technology and applications platforms in its products. This paper presents the business reasoning for reuse of software and processes in a product-line/platform vs. dedicated to a product and attempts to fully understand the reasoning and economics behind making such a decision.

Practice has shown that many inhibitors lie in the realm of management and decision-making. The most common causes of these failures are: resource constraints, lack of incentive, single-project view, time constraints, lack of clarity on reuse utility, and lack of education. Software project managers and developers need to achieve better understanding, estimation, evaluation, and quantification of the software reuse and associated business factors as well as their predictive relationship to software effort and quality.

Contemporary software reference models for reuse do not consider many of the technical and non-technical factors in their quantitative models. To mitigate this, the paper also explores the broader, industrial engineering perspective and its concepts of product platforms, product lines and other relevant methodologies to proposes strategies for

Abraham P., Sikka V. and Simpson G. (2010).
SHARE VS. OWN - Software Reuse using Product Platforms .
In *Proceedings of the International Conference on e-Business*, pages 245-255
DOI: 10.5220/0003006302450255
Copyright © SciTePress

designing and building reusable application software components.

We present a rigorous, cost benefit analysis based methodology for the evaluation of well-defined metrics to measure the benefits of a particular strategy and the associated costs. We recognize that the decision-making is essentially probabilistic, leveraging imperfect data, making such decision-making shades of gray among alternatives rather than black and white. The presented methodology accounts for these imperfections in the analysis.

Guidelines are developed to help a decision-maker decide when the long-term benefits involved in implementing and maintaining reusable coding procedures outweigh the short-term benefits of a dedicated implementation. Specific recommend-ations are made for coding practices, software design, documentation and management procedures that encourage and result in successful code reuse practices.

2 SOFTWARE REUSE INVESTMENT SUCCESS FACTORS

The industrial (manufacturing) world has been successful for many years in implementing a product-line approach to reuse using pre-fabricated (pre-manufactured, interchangeable) components. Product-line is closely related to the concepts of horizontal and vertical reuse. Horizontal reuse provides generic reusable components that can support a variety of products. Vertical reuse focuses on developing the preferred parts supporting a given family of related products or product-line. It is regular practice for these industries to assemble parts into products and use the same parts in more than one product within a "product-line" family.

Can this same "manufacturing" approach be used in software engineering?

SAP has certainly embraced parts of this philosophy with its Business Process, Application and Technology Platform strategies that serve as the foundation on which our *Business Suite* (Suite) and *Business By Design* (BYD) products are built.

2.1 Managing Diverse Software Products

To run any software component requires the use of other software and hardware artefacts that may be owned by SAP or shared among its partner ecosystem. *It is reiterated that, for this paper, the topic of reuse also includes artefacts used in the run time for the software components.* Assuming we have the relevant tools, processes and technologies for effective software component reuse, the evaluation of whether to reuse a software component (share) or develop it (own) is dependent on the particular reuse strategy we adopt.

There are two dimensions to the strategy.

- On one hand, we look at the product platform strategy and decide where a particular component should reside: in which architectural layer, using what technology etc.

- On the other hand, we look at the time horizon for the business initiative the software component is in support of: i.e. today's business, the next generation of emerging businesses, and the longer-term options out of which the next generation of businesses will arise.

At SAP, we have adopted multiple strategies depending on the nature, size, location and technology associated with the software component being reused. We briefly outline the problem environment in the graphic below:

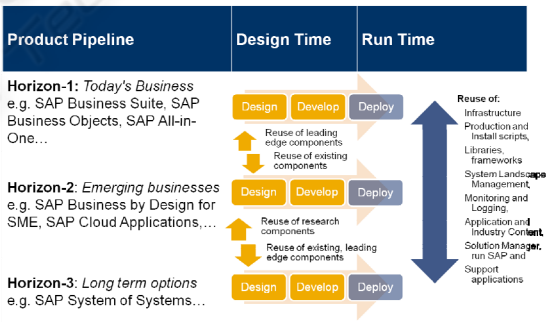

Figure 1: Shared vs. Own problem environment.

Note that there are two cycles in play. Across the functional cycle of design, develop, deploy, components change based on fit to purpose with upgrade and rebuild as the scope of the purpose changes over time. The vertical cycle is a much longer and slower moving one based on use density. As components are more frequently reused, they sediment down through the platform layers. This sedimentation can often include 3rd party infrastructure layers, who adopt technical capabilities initially developed for a single application solution. In the most effective model the

reverse flow is also managed, with components being factored out of the platform and back into products as the reuse decreases over time. This allows for the complexity and overhead of the platforms to remain optimal over time.

2.2 Balancing Commonality and Distinctiveness

At a fundamental level, product variety and fit is valuable in the marketplace. The need for superior performance of the products and the desire to preserve distinctiveness (e.g. custom features, control etc.) promotes product organizations to own certain key components. On the other hand, it is costly to deliver, as cost benefits are driven through commonality. The balanced sharing of assets across products allows companies to manage this trade-off.

This balance has however, temporarily resulted in an unwanted side effect in SAP: total cost of ownership (TCO) increases due to the complex configuration that we provide customers to tailor our software products to their needs. As mentioned above, parameterization is a valuable tool in leveraging shared assets to fit different solutions. However, the current architecture leverages the same parameterization for both SAP engineering product fit and on-premise customer fit (customization). The effect is to trade off customer complexity for the power of reusing, and therefore only having to support, a single mechanism.

Changing the product architecture can influence the nature of the trade-off. For example by the use of pre-configured and interchangeable software component for a particular industry vertical or customer group which hide the complexity of configuration will lower costs of customization. Another technology solution is the use of model-based methodologies that lower the fixed cost of developing software, and/or delivering the software as a service. The hypothesis is that this type of reuse promotes mass-customization, shortens the time to market and promotes consistency in products.

2.3 Common Architecture Strategy

The sharing or owning of software components is dependent on the architectural strategy. The architecture relates software components to a physical problem space (hardware, operating system, and application packages such as database or user interface). A common architecture lessens the need to make reusable software components highly generic because the environment in which they will be used is well defined. The architecture defines the rules for developing software components and provides standard interfaces and data formats. This aids in the inter-changeability of reusable software components across the product-line.

SAP had elements of a common architectural strategy from its inception. SAP uses this approach for the lower level technological platform and to support the user interface. However, there is considerable difference in higher layers of the architecture between our Business Suite and Business by Design (BYD) products, which leverage the same technology platform but are targeted at different markets. It should be reiterated that the platforms should be different if they are fundamentally different and that the determination of that “fundamental difference” is at the heart of SAP’s challenges.

2.4 Product Platform Strategy

Product platform strategy is the foundation of the existing SAP product strategy, which has multiple products related by common technology platform. It defines the cost structure, capabilities, and differentiation of the resulting products. When the market and products were less diverse, and the technology considerations more unified, separating product platform strategy from product line and individual product strategy allowed SAP to concentrate on its most important strategic issues of reliability and scale. As the diversity and rate of change has increased, the question as to which components products share and which are dedicated has ultimately tied to the product platform strategy more closely to the product line.

2.5 What is a Software Product Line?

A *software product line* is a set of software-intensive systems, satisfying the specific needs of a particular market segment, that share a common, managed set of capabilities and that are developed using a common methodology and leveraging common skills sets.

This definition is consistent with the traditional product line definition. But it adds more: it puts constraints on the way in which the systems in a software product line are developed. Substantial production economies are achieved when the systems in a software product line are consistently developed from a common set of assets in contrast to being developed separately, from scratch, or in an arbitrary fashion. It is exactly these production

economies that make the software product line approach attractive.

Production is made more economical when each product is primarily formed from existing components, tailored as necessary through pre-planned variation mechanisms such as parameterization or inheritance, adding any new components only when necessary, and assembling the collection according to the rules of a common, product-line-wide architecture. Building a new product (system) becomes more a matter of assembly than one of creation; the predominant activity is integration rather than programming. For each software product line, there is a predefined guide or plan that specifies the exact product-building approach.

Software product lines give *economies of scope*, which means that we take economic advantage of the fact that many of our products are very similar– not by accident, but because we planned it that way. We make deliberate, strategic decisions and are systematic in effecting those decisions. This concept must be contrasted with the specifics of a product platform, which is described next.

3 THE PRODUCT PLATFORM

We define a product platform as a collection of core assets that are shared by a set of products. These assets can be divided into four categories:

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

Processes and infrastructure - used to make or to assemble software components into products

Knowledge base – design know-how, mathematical models, testing methods and data sets

People and relationships – teams, relationships, between members and between teams

In certain manufacturing systems, these process and systems are themselves machinery like assembly lines or manufacturing centres etc. In the software arena new ideas like *software factory* embody this principle. Most companies do have parts of this automated with production and installation scripts, configuration of system landscapes etc. However, a coherent methodology and infrastructure is yet to emerge. The organizational aspects also need to be facilitated by automated systems much like manufacturing centres. This encompasses the knowledge base since a large part of the knowledge resides in people. Web 2.0 holds a lot of promise in this area and is being integrated into development and production tooling.

A product platform is primarily a definition for planning, decision-making, and strategic thinking. A product platform is not a product; it is a collection of the common elements, especially the underlying defining technology, implemented across a range of products. So in a sense this definition is broad, a generalization of the concept in SAP where we have of a technology platform and application platform but in another sense it is distinct, as it results in a collection of common elements. These common elements need not necessarily be complete in the sense that they are something that could be sold to a customer.

SAP markets and builds its products (Suite and BYD) as platforms for running the business processes of large enterprises (LE) and small to medium enterprises (SME) respectively. The defining technologies used to implement this business process platform will evolve over time, at different velocities and hence it is imperative to manage this effectively. The platform's unique differentiation provides a sustainable competitive advantage. Therefore it may be argued that all components that are related to business processes such as orchestration must necessarily be part of the platform and cannot be owned by an application or industry solution built on top of it.

3.1 The Platform Influence on SAP's Ecosystem

In the context of SAP's ecosystem, a platform may be viewed as a realization of the technology strategy that is made available through a set of access points or interfaces (APIs). Partner ecosystem members (ISV's and SI's) then leverage these interfaces as a kind of toolkit for building their own products and solutions, and think of them as the starting point for their own value creation. The platform is the "mechanism" through which the platform organizations share value with their ecosystem. Any product contains elements specific to a given use or solution and elements that are shared with many other products in the development ecosystem within and outside SAP development. The latter represent an opportunity that can be leveraged by other members of the ecosystem to eliminate redundant effort.

The architecture of products and services has a profound effect on the evolution of ecosystems. Well-managed platforms shape ecosystem dynamics as they grow to incorporate new functionality and create opportunities for SAP to expand its ecosystem. How a platform evolves and responds over time, shapes the ecosystem that depends on it: what firms survive, where diversity can exist, what will be easy to do and what will be hard, which things in the ecosystem will do well with little effort, and which things will be challenging. This happens because platforms serve as an intermediary between the underlying technology and the ways in which it can be easily exploited.

3.2 Platform Architectures

The choice of a defining technology as a platform strategy is perhaps the most critical strategic decision for a high-technology company. Typically, the defining technology of a platform differentiates the products that are based on that platform. While SAP is more defined by its business process centricity, the technology aspect of the platform strategy is still significant. Business applications can be classified in different operational or technical archetypes, based on their characteristics and requirements at run-time. A few examples of these archetypes are:

- **Online transaction processing systems (OLTP)**: characterized by low latency, high responsiveness, data integrity, predefined UI workflows. Instances of this archetype are e-commerce sites, CRM, e-banking systems.
- **Analysis systems or online analytic processing (OLAP):** characterized by their ability to produce complex analytical and highly customizable queries on large multidimensional datasets, with low latency responses. Business Intelligence (BI) systems fall into this category.
- **Batch systems**: capable of performing operations on large datasets efficiently, coordinating jobs to maximize CPU utilization and energy consumption with recovery policies when exceptions occur.
- **Networked systems**: software that integrates different applications and services into more complex solutions. It differentiates itself by delivering a business solution (e.g. Supply Chain Management) that manages the information and control flow across many other systems (Inventory Management, Order Processing).

SAP has products or components that cover all of these archetypes. Each of these application families has its own constraints, characteristics, and optimal design patterns that can be applied to solve the specific challenges they present. Very often, these challenges have conflicting goals. For example: OLTP will optimize for low latency, whereas latency for batch or networked systems is not as important. OLTP scales better horizontally and benefits from a stateless architecture, while batch systems scale vertically and tend to be stateful. The technical infrastructure and services to support each is consequently significantly different. The key point is that a platform's effectiveness is highly dependent on the archetype served. The more knowledge of the application a platform has, the greater its ability to increase the efficiency of running and operating it, and the greater the degree of sharing. Thus having multiple platforms that are tailor made to its constituency (by archetype, mode of delivery or size: LE, SME Micro) one could substantially lower the TCO.

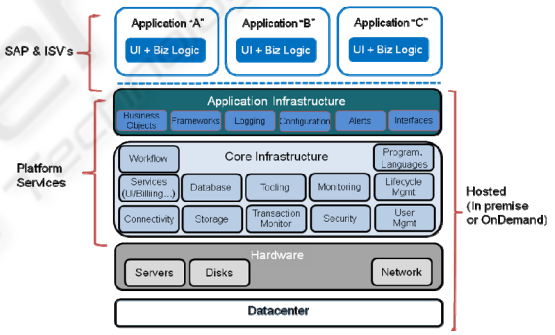

Figure 2: Increased reuse through an application runtime infrastructure.

An increased amount of shared components leads to higher levels of efficiency, so the question is which are the most natural candidates to be "extracted" from applications into the platform? The obvious candidates are those referred to application infrastructure services *viz*. application configuration, run-time exception, logging etc. Refer example above. Every application needs them, yet they are frequently written repeatedly for each platform.

By exposing these basic services publicly or by sharing them across platforms as libraries or frameworks, the platform has an increased ability to automate common procedures and offer more advanced operational management capabilities and lower the TCD by sharing the common platform

services. Thus, finer-grain tuning, customization and troubleshooting are made available. In the example above, notice that the hosting method does not need to understand in detail what the application does, but instead how it does it. (*e.g.*, where are connection strings to the database stored? How is run-time exceptions logged and notified?)

4 COST-BENEFIT ANALYSIS FRAMEWORK

Finally, given one or more credible product platform strategies for reuse, we would like to evaluate these alternatives to estimate the immediate economic benefits of reuse. The following CBA methodology is adapted from various publications and is presented for illustrating the benefits of a platform strategy and for completeness.

4.1 Metrics to Measure Benefits of Reuse

In this section, we present a conventional cost benefit analysis. Benefits of reuse are difficult to measure objectively and hence we present the current state of the art for it. Cost on the other hand is easier to compute and only a high-level view of what is pertinent pointed out. In addition, enterprises have developed their own methodology for cost accounting, which are uniform in spirit but not in detail. Software metric is any measurement that relates to a software system, process, or related documentation. Metrics are distinguishing traits, characteristics, or attributes that are both static and dynamic. The reuse metrics mentioned below are those developed by research teams at George Mason University in Fairfax, Virginia (see Rine & Nada 1998). These relate to the benefits of software reuse in the cost benefit analysis calculations that is introduced further down the paper. We also use the term module in place of software component as the basic unit for applying metrics. The distinction is somewhat academic but a software component may be too coarse grained for meaningful measurements.

In order for a metric value to be statistically valid, it is necessary to have a reasonable quantity of data. This data collection is unlikely to be successful unless it is automated and integrated into the development process. Finally, product data should be kept as an organizational asset and historical records of all projects should be maintained. Once an appropriate data set is available, model evaluation involves identifying the parameters that are to be included in the model and calibrating these using existing data. Such model development, if it is to be trusted, requires significant experience in statistical techniques.

The software reuse metrics are grouped into five major categories: *general, quality, parameterization, coupling, and cohesion,* which are also software engineering principles that correspond to the reuse attributes. The general category is for attributes that are not in the four software engineering categories. The following table lists the popular reuse metrics:

Table 1: Metrics to Mesure benefits of Reuse.

Category	*Metric*	Measures
General Metrics	*Time to market*	Reduction in development time
	New Product Opportunity Opportunities	Potential for long tail solutions etc.
	Understandability	Ease of adoption
	Size	Extent of the module
	Type of module	e.g. Specification, functional
Quality Metrics	*Consistency*	for centralized maintenance
	Ease of Change	The degree to which it can be changed
	Comments	Usefulness, understandable. accuracy
	Formatting	Readability of the code
Parameter-ization	*Functional Data*	
Coupling Metrics	*Coupling System External Coupling*	the strength of the interconnection and dependency among modules
Cohesion Metrics	*Functional cohesion*	degree to which each part of the module is necessary for performing a single function.
	Data cohesion	degree to which it has a single-data type associated with it

4.2 Cost Calculations

The methodology proposed here is an amalgamation of methodology proposed by the Software Institute in HP Labs and the Software Engineering Institute in CMU (see Clements et al., 2005, Petersen, 2004 and Malan & Wentzel, 1993).

4.2.1 Development Costs

Setup and Overhead. New systems will be needed to support a full-fledged systematic reuse program. It includes ongoing costs of expanding and maintaining reuse layer or system, a management support structure to ensure systematic reuse, and training programs, and should be assessed as indirect overhead.

Producer. The reusability of software components depends on a number of factors such as the degree of generality, complexity, and fit to expected use, as well as the quality of the component, and the extent and utility of documentation and accompanying test suites. Further, the component has to be available, and hence must be certified and entered in a platform, library, as software services or broadcast by some other means. Therefore, component producers face additional costs over and above the usual development-cycle costs, and these are estimated to be anywhere from 30% to 200% higher than the cost of producing a component not intended for reuse. This is true even for a component that is re-engineered from existing code.

Consumer. Selection, specialization and integration for reuse entails articulation of the component requirements in a suitable form, search and retrieval of the component, understanding of what the component does, and verification that it does indeed fit the purpose. The component may need to be specialized to fit the consumer's current needs. This involves adaptation (with co-requisite program understanding and subsequent testing). Lastly, the component must be integrated into the system under development, and tested.

Lifecycle Costs. The view of maintainability as a form of reusability is novel and important. It captures the idea of reusability in time within a dynamically evolving system. Evolutionary dynamic systems require reusability in time of unchanging parts of the system while other parts of the system evolve. By centrally maintaining the reuse components, managing their evolution, and propagating upgrades to new products as well as updated versions of older products, the organization can exploit further opportunities to reduce duplication of effort. Moreover, centralized enhancements to black box components enable a whole platform of derivative products to be produced more quickly at lower cost.

4.2.2 Probabilistic Nature of Calculating Cost

Time value of Money. When the reuse instances are expected to occur over a longer time horizon, the timing of the cash flows should be taken into account. This is done by incorporating a standard present value analysis into the model. One typically uses the Discount Cost Function (DCF) analysis. Typically Horizon-1 (today's businesses) components described previously fall into this category. In case of Horizon-2 & 3 (emerging businesses, and the longer term options) components, more market based approach such as real options valuation (ROV) may be used to account for the high uncertainty.

Uncertainty in Reuse Instances. The degree of uncertainty about the evolution of a product family tends to increase as the time horizon is stretched. Thus, anticipated reuse opportunities arising from products or upgrades planned in a multiyear horizon is likely to be much more uncertain than those in the current one-year business plan. To incorporate the uncertainty as to whether the component will indeed be reused, the probability of each reuse instance should be estimated, and the expected consumer savings computed. This is essentially a DCF calculation of the NPV and optionally decision tree analysis .

Future upgrades of Components. The maintenance and management of evolving components increases the cost to the producer/maintenance group. Consumers of the component benefit from not having to duplicate corrective and evolutionary maintenance activities, though they do have to incur some cost to incorporate upgraded component(s) into their products. This also is essentially a DCF calculation of the net present value (NPV).

4.3 Cost Benefit Analysis

It is clear from the discussion above that for some components the choice of strategy dictates whether it is shared or dedicated for a particular product. In many cases however software components may not have a predestined position in the architecture or is not obvious and hence a cost benefit analysis of shared versus own using historical data on similar projects should be performed. Since reuse involves multiple products evolving through their respective life-cycles, an assessment of the economic impact of a systematic reuse program must incorporate cost and revenue projections that extend beyond that of a single development project. A template such a cost

benefit analysis that should be performed for each component based on the research from the Software Engineering Institute (SEI) at CMU for is presented below.

Table 2: Cost functions used to compare building a reuse platform versus building stovepipe products

Function	Output
$C_{org}()$	Cost to setup and run an organization to adopt the product line approach for its products
$C_{cab}()$	Cost to develop a core asset base suited to satisfy a particular scope
$C_{unique}()$	Cost to develop the unique parts (both software and non-software) of a product that are not based on core assets
$C_{reuse}()$	Costs to build a product reusing core assets from a core asset base
$C_{prod}()$	Cost of building a product in a stand-alone fashion. It relies on historical data or general software engineering cost models for its evaluation.

Note: We assume that these functions accommodate influencing factors such as the time value of money, uncertainty in reuse instances and the probability future upgrades. Specific formula's are available in the research from several institutions like SEI, HP Labs etc. (see citations).

This cost can be expressed by Equation 1.

Cost of building a product line =

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

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. The cost of building *n* products independently, is expressed in Equation 2.

Cost of building *n* stovepipe products =

$$\sum_{i=1}^{n}(C_{prod}(product_i)) \quad (2)$$

4.3.1 Evolution and Upgrade

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, 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. One might 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 considered unique but now can be refactored into commonalities.

4.3.2 Benefits Calculation

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. However, the metrics discussed previously help one establish a list of benefits (= *nbrBenefits*) to be factored in the analysis as given in Equation 3.

Benefits of building *n* products using a product platform approach =

$$\sum_{j=1}^{nbrBenefits}(B_{ben_j}(t)) \quad (3)$$

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.

The contributions of the benefits are summed and then used to build a model equation as needed. For example, to express the development cost savings (or loss) from using the product platform approach as opposed to one-off development for each product equals [Equation 2] – [Equation 1]. A more complete picture of the cost benefit of using a product platform approach adds Equation 3 to that result.

4.3.3 Illustrative Examples of Reuse in the Design Time

The following example illustrates the impact of the approach in a new product line. The new product

line is targeted for the SME space and consists of about five components called distribution units (DU). Since this product line was targeted for a business user and in an effort to maintain the look and feel across product lines, it was decided that the current technology platform use for LE space called NetWeaver is the appropriate platform for this product line. However, experience has shown that there was potential to reuse a lot of features and services among the five DU's. Thus it was decided another (sixth) component called the application platform was to be developed by a producer group for this product line.

Challenge. Beyond building the initial business case, there is normally a low confidence level in any data related to the future success and adoption timeframe of a new software product line. This tends to drive the design decision making away from reusable/platform towards single use components - "get the first product out of the door and worry later". This can have potentially damaging effects on the actual ability to grow the product line in a cost effective manner but has always required subjective judgement on the part of the solution manager. Use of the CBA would allow the initial solution managers to "run the numbers" associated with making certain "reasonable estimates" around different componentization strategies and the short, medium and long term financial impact.

Solution. Based on measured data obtained from many prior software development projects, estimates show that with a 50% reuse level and a 5x quality improvement in the reused component over new code. However, Producer effort was increased by 108% and consumer effort reduced by 40% during the development phase. During the maintenance phase it is estimated that producer effort was increased by 25%. and consumer effort reduced by 42%.

These effort factors (assuming a 50% reuse level) are used together with the following assumptions to estimate reuse benefits:

- Hourly rate for software engineers (including basic salary $75 and administration overhead)
- Project team size 20
- Development cycle time without reuse (months) 12
- Annual inflation in labour rate 5%

A simplifying assumption that all of the products are comprised of the same amount of new and reused code is made to better demonstrate, a number of points. We have also simplified the cost benefit calculations and used only NPV calculations. We divide the calculations into four models that have self-explanatory titles. The model results are shown in the table below.

Table 3: Examples of Reuse in the design time.

Model 1: Basic Development Phase Costs

	Producer Cost	Product1	Product2	Product3	Product4	Product5
Year of Release	0	1	1	2	2	3
Without Reuse	300,000	630,000	630,000	661,500	661,500	694,575
With Reuse	624,000	378,000	378,000	396,900	396,900	416,745
Reuse specific Overhead		35,000		25,000		25,000
Consumer Saving		252,000	252,000	264,600	264,600	277,830
Cumulative Net Saving	(-)624,000	(-)407,000	(-)155,000	84,600	349,200	602,030

Model 2: Taking the Time Value of Money Into Account

Consumer Saving after Discounting [i]		234,419	234,419	228,967	228,967	223,642
Cumulative Discounted Net Saving	(-)624,000	(-)422,140	(-)187,721	19,613	248,580	452,098

Model 3: Taking Uncertainty In Reuse Instances Into Account

Probability of Reuse		1	1	0.90	0.75	0.50
Consumer Saving with Discounting & Uncertainty		234,419	234,419	206,070	171,725	111,821
Cumulative Discounted Expected Net Saving	624,000	422,140	187,721	3,284	168,441	260,138

[i] The interest rate may be the prevailing bank rate, reflecting the interest that the investment would earn if it was deposited instead of invested in reuse, or the company's hurdle rate, reflecting what the investment would earn in some alternative use within the company.

Table 4: Examples of Reuse in the design time. (cont.)

Model 4: Including a Future Upgrade

	Producer Cost	Product1	Product2	Product3	Product4	Product5
Year of Release	1	2	2	3	3	4
Probability of Reuse		1	1	0.68[ii]	0.38	0.25
Upgrade without Reuse	75,000	157,500	157,500	165,375	165,375	173,644
Upgrade with Reuse	93,750	91,350	91,350	95,918	95,918	100,713
Reuse specific Overhead		10,000		10,000		10,000
Additional Consumer Saving		57242	57242			
				38019	21246	13653
Cumulative Discounted Expected Net Saving	(-)717,750	(-)467,301	–175,641[iii]	38,766	231,737	329,599

5 CONCLUDING REMARKS

SAP, the world's largest provider of enterprise applications software, originally architected its reuse strategy around horizontal application and technology platforms that provided focus on the scaling and reliability desired by its mainly homogenous enterprise market. On top of this strategy, it built a large and geographically distributed organization and a large portfolio of diverse products.

Recently, an increased rate of change in market needs and supporting technology innovations has stressed that strategy. Solution managers, development decision makers, are challenged to effectively handle the conflicts of rapid solution delivery while identifying candidate components for application or infrastructure reuse. This complexity extends out beyond the company into its ecosystem of partners and customers as they fit the applications to specific business needs. Thus, guidelines should be developed and specific recommendations made to streamline this evaluation process.

It is our position that a product-line approach, supported by provable cost benefit analysis, is a more effective model for delivering reuse benefits in this dynamic market environment. As software industry models of reuse are not sufficiently robust, we have looked to traditional manufacturing industries for guidance; moulding their models to fit the imperfect data base of software decision making.

This paper proposes a cost benefit analysis based model which, when combined with a methodology, software engineering tooling and organizational guidelines, will enable the engineering management to effectively balance product specific and platform reuse requirements, in a cost and market effective manner.

As this model has not yet been adopted, the paper also describes the steps necessary to fit the proposed approach to a specific organization and how the calculus would provide objective componentization and reuse data. Solution management in the design of new product lines would leverage this. Comparative examples are given covering the first product in a new line, new market segment and the first product in a product line largely similar to an existing line. These two scenarios have significantly different subjective influences, requiring different use of the cost/benefit analysis.

REFERENCES

Clements, Paul C., McGregor, John D., Cohen, Sholom G., 2005. *The Structured Intuitive Model for Product Line Economics (SIMPLE)*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Dale R. Peterson, F. van der Linden (Ed.), 2004. *Economics of Software Product Lines,* PFE 2003, LNCS 3014, pp. 381–402, 2004. Springer-Verlag Berlin Heidelberg.

Iansiti, Marco and Levien, Roy., 2004. *The Keystone Advantage: What the New Dynamics of Business Ecosystems Mean for Strategy, Innovation, and Sustainability*, Harvard Business Press.

Malan, Ruth, Wentzel, Kevin, 1993. *Economics of Software Reuse Revisited*, Software Technology Laboratory.

[ii] Conditional on Product 3 being produced, the probability of its upgrade being produced is assessed as 0.75. Thus, the probability of producing the upgrade is 0.9*0.75 = 0.68. Similarly, the conditional probability of upgrades to products 4 and 5 being produced is 0.5.

[iii] This is the cumulative saving for the first product plus its upgrade. Overhead for the year is assumed to be incurred regardless of whether the next reuse instance is actualized (i.e. a semi-fixed cost, such as salaries).

McGrath, Michael E. 2001. *Product Strategy for High-Technology Companies: Accelerating Your Business to Web Speed*, McGraw-Hill.

Rine, David C., Nada, Nader 1998. *Software Reuse Manufacturing Reference Model: Development and Validation*, School of Information Technology and Engineering, George Mason University, Fairfax, Virginia.

Robertson, David, Ulrich, 1998. Karl *Planning for Product Platforms*, Sloan Management Review.

Viguerie, Patrick, Smit, Sven and Baghai, Mehrdad, 2008. *The Granularity of Growth: How To Identify The Sources Of Growth And Drive Enduring Company Performance*, John Wiley & Sons (US).