Issues and Models in Software Product Lines

Jorge L. Díaz-Herrera
Department of Computer Science, Southern Polytechnic State University (SPSU)

Peter Knauber
Fraunhofer Institute for Experimental Software Engineering (IESE)

Giancarlo Succi
Department of Electrical and Software Engineering, University of Alberta

Abstract

Software product lines are one of the most promising fields in software engineering. They aim at the synergistic construction of software products.

A successful introduction of software product lines requires three essential ingredients: a business analysis of the overall advantages that can come from product lines, the definition of a systematic process for product lines development, and the definition of general models, in a standard format, which can guide the development process.

1. Introduction

Software product lines are one of today hottest proposals to improve the effectiveness and the efficiency of software production.

The aim of software product lines is to produce multiple products –often in a given domain, trying to exploit scope economies. Scope economies arise when the cost to develop multiple products or deliver multiple services together is less than the sum of the cost of developing the products or delivering the services individually. Scope economies are common in knowledge intensive sectors (Baumol *et al.*, 1982; Withey 96). The underlying assumption is that the benefits from the reuse of domain specific software components will offset the extra cost for the increased organizational complexity.

Software product lines are appealing; some research has been already performed in the subject, such as (Bass *et al.*, 1999; DeBaud and Knauber, 1998, Rhine and Sonnemann, 1998). Reuse has been considered the cornerstone of software product lines –see for instance the very clear paper of Poulin (1997). Sometimes additional factors have been taken into account, such as in (Clemens, 1999); in most of the cases, they were mainly impeding factors.

Several discussions have taken place on software product lines. Three issues appear especially important for a sound product line practice. An appropriate business analysis of the advantages of software product lines determines the overall feasibility of a product line initiative and the area where to focus. A well-defined and and comprehensive process tailored to the respective development environment is an important factor to succeed in the software business in general; it becomes essential for product line development, where the complexity of the problem explodes, given the parallel and synergetic development of multiple products sharing the same organizational resources. Models and standards are the mediums by which the coordination among different people and products in the line occur.

In this paper we present a comprehensive view of software product lines, by analyzing these three issues and proposing suitable models for them. In section 2 we focus on the business analysis, in section 3 on the definition of the process, and in section 4 on the coordination with models and standards. In section 5 we present come conclusions and lines for future research.

2. Business analysis for software product lines

Product lines are not a silver bullet, nor their advantages simply happen. They require careful investment and smart strategies. In particular, it is important to take advantages of phenomena and technologies that support the establishment of profitable product lines, such as: branding, network externalities, minimal marginal costs of production, sharing of organizational costs, software reuse, and modularity.

Branding arises when people prefer a brand they have experienced in the past based on expectations they have on the future behaviour of such brand (Marder, 1997). Branding is especially strong in software, where the expectations of quality, reliability, and usability of a product are usually not met. Branding supports the establishment of a product line: users will feel more comfortable in buying a product of a firm or a consortium if they are already satisfied by another product with a similar brand from the same firm or consortium or from a firm or consortium endorsing the new product.

The *marginal cost* to produce an extra unit of a software product is *minimal*. Therefore, producers can segment the market freely, with the only constraint of opportunity costs. This can be profitably used dealing with network externalities.

Network externalities occur when users value a product more when there are other users of such product or of other “compatible” products (Economides, 1996). Software is an industry with strong network externalities coming from the need to exchange information between users and products and the advantage of exchanging experience among users of a product.

Developers of product lines can organize the cross-compatibility of the different products to maximize network externalities; users of a product in a product line will achieve much higher benefit buying another product in the same product line rather than a product from competitors, ceteribus paribus. In addition, the minimal marginal cost of software products can be profitably used to enlarge network externalities. Product lines developers can provide a product in the product line free or at a substantial discount creating value for the other, fully priced products in the product line.

Sharing organizational costs refers to possible positive synergies in marketing and distributing the software products in the line. In marketing, while advertising the feature of one product of the line it is possible to evidence the need for another product in the line. In distribution, it is possible to take advantage of the distribution channels of one product of the line to distribute also the other product.

Software reuse is the systematic use of prefabricated software components in new products. A product line offers the opportunity of identifying common components in the line, thus fully exploiting the advantages of software reuse. Often product line reuse is based on domain-centered software architectures (DeBaud *et al.*, 1998; Bass *et al.*, 1998).

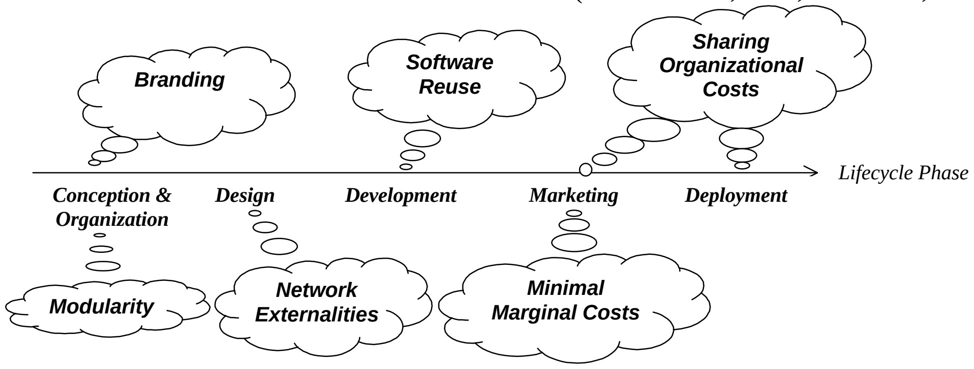

Figure 1: The possible advantages of software product lines

Modularity is the attempt of reducing the complexity of the production by organizing it into separate, cohesive units that relate one-another according to a well-defined protocol (Simos, 1982). Modularity can be applied to the product, that is the software system being developed –this is the essence not only of “Modular Programming” but also of most of the proposed programming paradigms, including object orientation, logic programming, and functional programming. Modularity can be applied to the development process itself. This is particularly useful in software product lines where the complexity of production tend to be very large.

From the analysis of the phenomena and technologies it is evident that there are two orders of scope economies that can come from product lines:

- Savings coming from the added value that products in the product lines have for users with respect to individual products
- Savings coming from distributing some of the fixed costs of components across multiple products; this include synergetic marketing (Churchill and Peter, 1998), sharing distribution channels, and software reuse

These scope economies are present throughout the development cycle. Figure 1 presents a summary of this view in a very simplified software product line lifecycle.

Taking advantage of these phenomena and technologies may require expensive up-front and operational costs. This is why, deciding on the production of product lines it is important to determine where the savings are going to be most relevant for the target product line and then studying whether such savings are likely to require feasible investments.

A careful analysis of the strategic positioning of the firm is required, especially with reference to the supporting phenomena and technologies. There are seven factors requiring specific attention.

The first factor is *the structure of the market*. This means: **(1)** whether the market they plan to enter is a natural monopoly, is an oligopoly, or is an open market (Baumol et al., 1992); **(2)** the number, the mutual positions, and the wealth of the incumbents; **(3)** the compatibility and interoperability of the products of the incumbents; **(4)** the relations between the target market and other lateral markets in terms of compatibilities or complementarities of products. This understanding is important for introducing any product, but is critical for product lines where the goal is to achieve a synergy from the presence of multiple products. If the market is a natural monopoly with a very strong incumbent and very limited relations with other markets, there is little hope to succeed at all. Open markets with various relations with other markets are the best for product lines and offer opportunities for network externalities. In addition, the structure of the market influences the possibility of sharing organizational costs between products in the product line.

The second factor is the *position of the producer in the market*, that is, whether the producer is a new entrant, an already established incumbent or the market leader. This is essential to understand the possibility to take advantage of network externalities and branding effects. A new entrant in a market close to lateral markets is unlikely to take advantage of branding, but can still try to take advantage of externalities, if some of the incumbents have products that are open to or interoperable with compatible products. An incumbent in a market open to later markets can take full advantage of both branding and network externalities to establish a software product line.

The *financial situation of the producer*, the *reliability of the predictions of the future users' base* and the *expected variability in the application domain* are three key factors in determining the scope of the strategy and consequently the investment to make. Producers are not advised to perform large up-front investments, such as establishing a domain-oriented library of reusable components, when

1. they are short of funds, or
2. the target markets do not offer reliable predictions of the future users' base, or
3. the products are likely to change a lot without a predictable variation pattern –for instance, for a high rate of innovation of the target domain.

On the contrary, when producers can invest money in uncertain operations, when the market is fairly stable, or when the variability of the application domain is predictable it could be wise to establish a fully-fledged reuse program.

In addition, the financial situation of the producers and the structure of the users' base influence strongly how to take advantage of minimal marginal costs. The structure of the users' base helps determining the possibility of sharing organizational costs.

The *technical competence of the producer* is the sixth factor to consider, especially if a new product line initiative coupled with the introduction of new technologies, such as object orientation. It would be risky to introduce object orientation in a business environment with experienced Cobol programmers, without an up-front commitment to devote time and effort to training. In addition it is important to analyze it before deciding to adopt any business process reengineering or improvement strategy, such as the adoption of a modular development process.

The *quality of the process of the producer* is the seventh factor to review when deciding to adopt a modular production process, as a software product line.

	Go/ Stop	Branding	Network Externalities	Marginal Cost	Sharing Org. Costs	Software Reuse	Modularity
Structure of the market	X		X		X		
Position of the producer in the market		X	X				
Financial situation of the producer				X		X	
Structure of the users' base				X	X	X	
Expected variability in the domain						X	
Technical competence of the producers						X	
Quality of the process of the producer						X	X

Table 1: Criteria for a Business analysis of software product lines

Table 1 contains a summary of the discussion so far. The rows contain the major factors to consider before establishing a given product line. Columns 3 to 8 contain the major phenomena and technologies that help establishing a product line. Column 2 refers to the strategic decision of whether or not to start a software product line. An X in a cell indicates that the corresponding factor should be considered when deciding on taking advantage of the corresponding phenomena / technology.

As table 1 implies, it is not wise to try to use all the phenomena and technologies to establish one single software product line. Producers' strategists should careful select only the few essential ones

3. A method for product line software development

Some factors make the development of software in product lines even more complex than one-at-a-time development of software systems. One of the main challenges to handle are the organizational changes that have to go hand in hand with the switch to product line development, other problems are the handling of shared resources and the production of shared assets within more than one (parallel) project. This section sketches the main organizational and process challenges to address for product line development and a method which can manage these challenges.

To deal successfully with these and other arising issues, a well-defined development process is essential. That process has to support all product line lifecycle phases, that are,

- construction of the product line infrastructure (encompassing business analysis to identify and define an appropriate scope for the product line (compare section 2), creation of a product line (or domain) model, and development of a reference architecture for all products in the line),
- usage of the infrastructure to systematically derive products (that is, product line instances) for specific customers, as well as
- evolution of the product line infrastructure as the domain evolves and matures over time.

Experience from projects proved that especially the introduction of a product line approach into an existing development organization is critical, because several important factors have to be taken into account and have to change according to the new development paradigm. Some of them are listed below:

- *The structure of the development organization.* The construction of the product line infrastructure is not a project-specific task, as assets created during that phase will be reused within other projects. Nevertheless, their development has to take place during concrete customer projects (basically nobody can afford the initial investment and risk of developing the complete infrastructure upfront). To ensure the reusability of these assets, there has to be a person or a group with the responsibility and the authority to make sure that assets belonging to the product line infrastructure are developed in a reusable way (that is, not only specific for the current project).
- *The "culture" of the development team.* People have to be trained in and motivated for reuse, that is, the not-invented-here syndrome has to be avoided. Development for reuse and development with reuse has to be established in the minds of the development team.
- *The relation to customers.* There are two main issues to consider. First, customers should be influenced in the functionality they require in a way that there is as much overlap as possible with the functionality realized by the product line. Second, financial planning of projects should change in a way that customers no longer pay for a product (the one they get) but for a service: the one to produce a specific product (instance) from a product line infrastructure. (The analogy of car production can illustrate these two aspects: First, the variants of cars to be produced from the factory are planned beforehand and customers can order pre-planned car variants only; other tuning of cars is very expensive! Second, customers pay the price for the service of producing a car according to their needs; there is no customer-visible relation between the price of one specific car and the producer's investment of building the car factory.)

Not only the development organization has to change: Like for single system development there is no one-fits-all product line development process. Thus, in order to be successful, a product line method has to be tailorable to the specific characteristics of the existing environment. Examples of these characteristics are processes currently established, notations used for documentation, programming paradigms (as well as programming languages), tools used for the development, and configuration management practices.

To address and manage the challenges of establishing product line development within organizations, the PuLSE™ (Product Line Software Engineering, see Bayer et al., 1999) method was developed at Fraunhofer IESE. The elements of this method, covering the whole product line lifecycle, are briefly described here. The method consists of three main elements: the deployment phases, the technical components, and the support components.

- The *deployment phases* are logical stages a product line goes through: Initialization of PuLSE, construction, usage, and evolution of the product line infrastructure. The phases describe the activities performed to set up, use, and evolve the product line.
- The *technical components* provide the technical know-how needed to operationalize the product line development. They are used throughout the deployment phases to cover the whole product line lifecycle, but can also be applied separately to address specific product line-related aspects.
- The *support components* package guidelines on how to introduce, deploy, and evolve product lines in specific contexts. They are used by the other PuLSE elements.

The technical components are described roughly in subsections 3.1 to 3.6.

3.1 Baselining and Customization: PuLSE-BC

PuLSE-BC's role is to instantiate PuLSE and to customize it to the specific environment of a development organization. In a baselining phase, factors characterizing the existing environment and its effects on product line introduction are elicited, then PuLSE is customized accordingly. Customization comprises the derivation of a complete process, including the definition of models (compare section 4) and workproducts used, their relations, and their representations depending on the elicited factors.

3.2 Scoping the Domain: PuLSE-Eco

PuLSE-Eco is the first of three components used in the construction phase of the product line infrastructure. Within PuLSE, PuLSE-Eco is used to determine an economically viable scope for the product line, depending on the business goals addressed. Moreover, it provides valuable information for distinguishing domain-related from application-specific components during the development of a product line reference architecture (see PuLSE-DSSA in Section 3.4).

3.3 Modeling the Product Line: PuLSE-CDA

In PuLSE-CDA, information about the product line as scoped by PuLSE-Eco is elicited and modeled using the workproducts and the process as defined by PuLSE-BC. The different, often domain-specific, workproducts (e.g., workflow diagrams, message sequence charts, data models) capture complementary views on the product line.

Populating the workproducts combines elicitation of information about single systems and consolidation of this knowledge in a generic product line model that captures variability. To derive the specification of a single product line member from that generic model, a domain decision model is created that contains a structured set of decisions corresponding to variability in the workproducts. To define a specific product, the decisions are resolved (see PuLSE-I in Section 3.5).

3.4 Developing a Reference Architecture: PuLSE-DSSA

PuLSE-DSSA supports the definition of a domain-specific software architecture, which covers current and future systems as described by the product line model produced by PuLSE-CDA. The architecture is descripted from different views (Kruchten, 1995) and may optionally be supported by a prototype of the architecture.

During the creation of the reference architecture, implementation-specific decisions are collected that will have to be resolved during architecture instantiation. These decisions and their possible resolutions are captured in the architecture decision model that extends the domain decision model. This architecture decision model supports the tailoring of the reference architecture for specific products, but it is also used to select among different implementations of a software component (interface), in cases where more than one implementation is possible.

3.5 Instantiating Product Line Members: PuLSE-I

PuLSE-I aims at specifying, instantiating, and validating single product line members. This encompasses the instantiation of the product line model and the reference architecture using the decision models from PuLSE-CDA and DSSA, the creation and/or reuse of products that constitute the instance, and validation of the resulting product.

3.6 Evolving and Managing the Product Line: PuLSE-EM

PuLSE-EM monitors and controls the development and evolution of the product line infrastructure and coordinates the activities of the other PuLSE components over time. It gets various workproducts from the different components, consolidates them, and takes care of their consistent evolution and maintenance, thus also encompassing configuration management tasks.

The previous sections briefly describe a first full version of the PuLSE™ method. Current descriptions of the method and its components are available under http://www.iese.fhg.de/pulse.

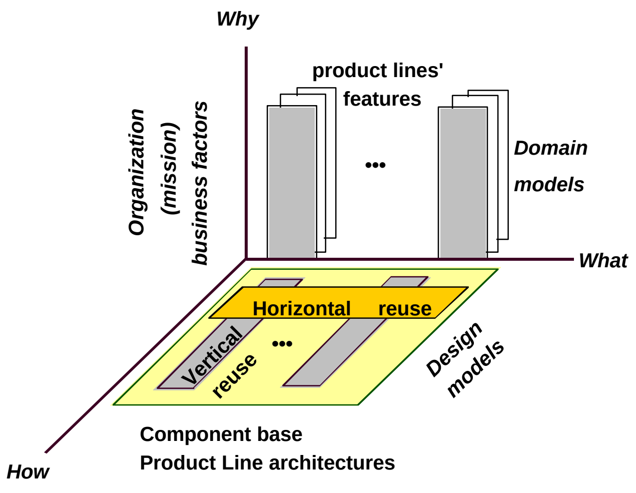

Figure 1: PL domain and solution models, an organizational view.

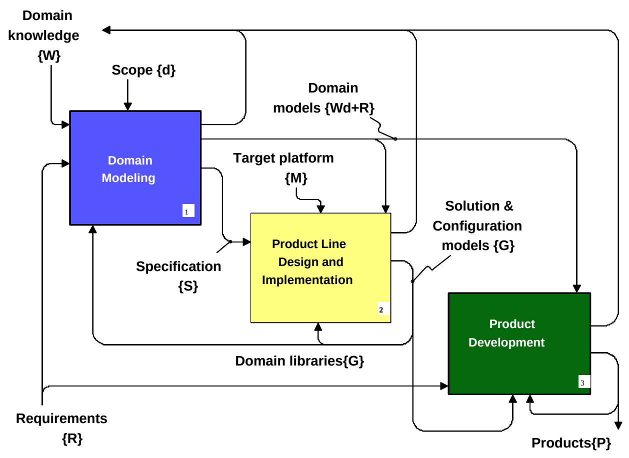

Figure 2: Domain modeling, PL design and implementation, and product development, a process view

4. Models and standards

In this section, we describe the technological aspects and standards needed for modeling and developing product lines. Due to space constraints, we do not discuss specific technology in detail. Rather, we suggest a roadmap for the various phases and kinds of artifacts produced. In general, we need modeling methods and specification languages, architectural styles, frameworks, and architectural description languages, as well as component metamodels, patterns, and component description languages. There are several kinds and types of models, which can be grouped into two large categories: specification models and design models. Specification models primarily include domain and requirement models, whereas design models include solution and configuration models. All these models then serve as the basis for requirements and design analysis for a specific product, and they support creation of component libraries.

These two sets of models are related from an organizational and from a process framework view (see Figures 1 and 2, respectively). To ensure higher probability of success, product lines are organized around ongoing business strategic goals or mission (Figure 1) as discussed in section 2. The established product line infrastructure, as discussed in section 3, integrates the corresponding modeling, planning, and asset construction activities necessary for systematic reuse. From this development viewpoint, there are three basic phases in the overall systematic reuse process as discussed earlier. In Figure 2 below, we illustrate the interrelations between these three phases. We use the formalism introduced by Gunter et al [GUN-00] in their reference model for requirements engineering to label the arrows.

4.1 Specification models

For product line development, specification models primarily involve Domain Models, but also include requirements models. A domain model formalizes knowledge about existing systems in a domain {W} encompassing known facts about the domain environment or outside *W*orld. They may also form the basis for business process reengineering. The requirements {R} specifies needs from an end-user viewpoint.

The specification model also contains descriptions of the various product lines in terms of mandatory and optional features, including functional and non-functional features. These models are represented using information modeling techniques such as object models (use cases are very useful here) and feature models.

The domain modeling activity is constrained by the chosen scope {d} thus sub-setting the domain knowledge to the specific set of products {Wd} based on strategic goals and organizational mission. The specification produced {S} is a precise description of {Wd}.

4.2 Design models

Design models describe the generic *solutions* that are the result of PL design and implementation. The Target Platform, or ***M***achine, {M} constraints the generic solutions {G} to the specific computing environment(s) on which the derived ***P***roducts {P} will execute. Solution models include PL architectures depicting the structure for the design of related products, and component designs specifying explicit control of variability.

Design models also include *configuration* models with specific information to support adaptation. Optional parts are very important. Possible combinations of optional features must be supported by the product line architectural design and the flexibility incorporated in the component design, otherwise it may be impossible to reuse an asset "as-is" because commonality and specificity are mixed. This would make it necessary to modify the asset when reused in the new context, and this should obviously be voided whenever feasible. Configuration models help here.

Solution models represent both software and hardware architectures suitable for solving typical problems in the domain. Solution models include the PL architecture and component designs.

- *Product line architectures* capture common high-level structures of related products (design-for-commonality) for building products, identifying common assets, and defining the means for connecting these assets. They allow integration of optional/alternative components (control-of-variability). Implementation decisions are outside the scope of the architectural design. They belong in the detailed design of the components and the connectors, which together form the component base.
- *The Component base* specifies common functionality across product lines with direct control of variability. There are different ways to control variability, e.g., class hierarchies, generators, generic parameters (templates), libraries, configurations, etc. Components, through design-for-commonality, implement those *crosscutting aspects* that are typically present in the majority of the product lines in a domain. Components also capture the way that these aspects may vary from one product to another, and provide plug-in compatibility via standard interfaces – control-of-variability.

Configuration models map between the problem models and solution models in terms of product construction rules, which translate capabilities into implementation components. These rules describe legal *feature* combinations, default settings, etc. For example, certain combinations of features may be not allowed; also, if a product does specify certain features, some reasonable defaults may be assumed and other defaults can be computed based on some other features. The configuration model isolates abstract requirements into specific configurations of components in a product line architecture.

4.3 Implementation issues

Models are subject to validation and verification before being basedlined. Model validation and verification as well as product validation and verification is carried out through a continuous cycle. One of the most critical questions is to be able to define correct, unambiguous and useful mappings between all these sets of conceptual elements and generated artifacts. For example, it follows that $S \subseteq R \subseteq Wd$, and that $S \Leftrightarrow M \wedge P$, where {W+R}= abstract problem model{S+M}=concrete problem model, G = generic solution model, and P = concrete solution model. A complete analysis of these equations is beyond the scope of our current discussions.

Currently, organizations are emphasizing *component-based development* (CBD) in the hope that this will bring about the elusive goal of systematic reuse. The idea has had certain amount of success in commercial products, such as OMG's CORBA, Microsoft's COM and Sun's JavaBeans [Leavens 2000], mainly due to "flexibility through a capability of composition" and not necessarily due to technological developments. The notions of vertical and horizontal reuse have been formally incorporated in important software construction technology such as CORBA. The top layers of the CORBA architecture specify standard objects that can be shared by applications across domains; they are known as the Horizontal CORBA Facilities. Standard objects that are reusable within products in a given product line are referred to as the Vertical (Domain) CORBA Facilities.

Object-oriented systems are typically monolithic pieces with difficult to detach "components." Current object-oriented component technology (including JavaBeans, ActiveX, COM, etc) imposes severe constraints on the components, tightly coupling them to implementation infrastructure. Truly reusable components must be specified as free of constraints as possible.

5. Conclusions

The establishment of a software product line is a complex operation that requires considerable more attention than just introducing a product, since there are complex interactions among products in the product lines and between products in the product lines and other incumbent.

Software Product Lines require a consolidation of understanding of the software systems in a given “domain” in order to *exploit commonality* and *anticipate and control diversity*. Analysis is different from conventional development. We must conduct a special, more comprehensive study of the applications in order to identify the features that are typically “present” in those applications, and particularly, the way that these vary from one application to another. That is, we must anticipate change and build adaptation capabilities into “standard” components.

The focus is thus one of analysis and design of variability within a set of products and the analysis and design of commonality across products. Design-for-Commonality forms the basis for standardizing assets to build products in the line by encapsulating the common features, and by defining a common architecture. Control-of-Variability is the basis for providing flexibility in the assets to meet requirements for specific products without compromising commonality; it requires careful design to include appropriate levels of parameterization, generalization and specialization, and extension.

Acknowledgements

The third author thanks the University of Alberta, the Canadian Natural Science and Engineering Council, and the Government of Alberta for partially sponsoring this research.

References

Bass, L., G. Campbell, P. Clements, L. Northrop, D. Smith. *Third Product Line Practice Workshop Report*, CMU/SEI-99-TR-003, March 1999 (This report can be found at the URL: http://www.sei.cmu.edu/pub/documents/99.reports/pdf/99tr003.pdf)

Bass, L., P. Clements, and R. Katzman. *Software Architecture in Practice*, Addison Wesley, 1998.

Baumol, W.J., J.C. Panzar, and R.D. Willig. *Contestable Markets and The Theory of Industry Structure*, Harcourt Brace Jovanovich, Inc., 1982.

Bayer J., O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, J.-M. DeBaud. *PuLSE: A Methodology to Develop Software Product Lines*, Symposium on Software Reliability (SSR’99), 1999

Churchill, G.A. and J.P. Peter. *Marketing – Creating Value for Customers*, 2nd Edition, Irwin McGraw-Hill, 1998

Clements, P. *Essential Product Line Practices*. Proceedings of the Ninth Annual Workshop on Institutionalizing Software Reuse (WISR 9), Austin, TX, January 1999 (This paper can be found at URL: http://www.umcs.maine.edu/~ftp/wisr/wisr9/final-papers/Clements.html)

DeBaud, J.M., O. Flege, and P. Knauber. *PuLSE-DSSA - A Method for the Development of Software Reference Architectures*. Proceedings of the 3rd International Workshop on Software Architecture (ISAW-3), Orlando, FL, November 1998

DeBaud, J.M., and P. Knauber. *Applying PuLSE for Software Product Line Development.* Proceedings of the European Reuse Workshop ’98, Madrid, E, November 1998

Economides, N., *The Economics of Networks*. International Journal of Industrial Organization, **16** (4) 1996

Gunter, C. A., Gunter, E. L., Jackson, M. and P. Zave. *A Reference Model for Requirements and Specifications*. IEEE Software, May/June 2000, pp 37-43.

Kruchten P., *The 4+1 View Model of Architecture*, IEEE Software, 1995

Leavens, G. T. and M. Sitaraman. *Foundations of Component-Based Systems*. Cambridge University Press. Cambridge, UK. 2000

Marder, E., *The Law of Choice*, Simon and Schuster Inc., 1997

Poulin, J. *Software Architectures, Product Lines, and DSSAs: Choosing the Appropriate Level of Abstraction*. Proceedings of the Eighth Annual Workshop on Institutionalizing Software Reuse (WISR 8), Ohio State University, Columbus, OH, March 1997 (This paper can be found at URL: http://www.umcs.maine.edu/~ftp/wisr/wisr8/papers/poulin/poulin.html).

Rhine D.C., R.M. Sonnemann. *Investment in reusable software. A study of software reuse investment factors,* The Journal of Systems and Software, Volume 41, 1998

Simos, H., *The Science of the Artificial*, The MIT Press, Cambridge, MA, 1982

Withey, J. *Investment Analysis of Software Assets for Product Lines*. CMU/SEI-96-TR-010. Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 1996.