

New Methods in Software Product Line Development

Charles W. Krueger

BigLever Software, Austin, TX

ckrueger@biglever.com

Abstract

A new generation of software product line success stories is being driven by a new generation of methods, tools and techniques. While early software product line case studies at the genesis of the field revealed some of the best software engineering improvement metrics seen in four decades, the latest generation of software product line success stories exhibit even greater improvements, extending benefits beyond product creation into maintenance and evolution, lowering the overall complexity of product line development, increasing the scalability of product line portfolios, and enabling organizations to make the transition to software product line practice with orders of magnitude less time, cost and effort. This paper describes some of the best methods from the industry's most recent software product line successes.

1. Introduction

The first generation of software product line case studies – from the 1980's and 1990's – described patterns of software development behavior that were characterized *a posteriori* – after the fact – as *software product line development*. Although these pioneering efforts were all motivated to improve productivity through software reuse, they were each independently discovering a special pattern of reuse that we now recognize as software product line development[1].

These fortuitous and unintentional reinventions of software product line practice are still recurring in the industry today. What is more interesting, however, is the current generation of *intentional* software product line initiatives, where organizations adopt with forethought the best practices discovered from the practical experiences of their predecessors. These cases provide insights about a new generation of software product line methods and the benefits that they offer over the first generation.

In this report, we describe three of the new methods that have provided some of the most significant advances to software product line practice, *software mass customization with configurators*, *minimally*

invasive transitions to software product line practice, and *bounded combinatorics*. The topics discussed here are based on firsthand experience with industry success stories at BigLever Software and other commercial software development organizations, including 2004 Software Product Line Hall of Fame inductee Salion and Engenio (a division of LSI Logic), elected nominee for the 2006 Software Product Line Hall of Fame[2,3].

2. Software Mass Customization

Application Engineering Considered Harmful

Software mass customization is a software product line development methodology distinguished by its predominate focus on domain engineering of reusable software assets and the use of fully automated production to virtually eliminate manual application engineering of the individual products. Analogous to the *mass customization* methodology prevalent in manufacturing today, software mass customization takes advantage of a collection of “parts” capable of being automatically composed and configured in different ways. Each product in the product line is manufactured by an automated production facility capable of composing and configuring the parts based on an abstract and formal *feature model* characterization of the product[4].

2.1. Sans Application Engineering

At the risk of abusing an well-worn computer science cliché, this section might best be entitled “application engineering considered harmful”. Application engineering, which is product-specific development, constituted a significant part of early software product line development theory, but has proved to be problematic in practice. One of the key contributions of software mass customization is to eliminate application engineering and the negative impact it has on software product line development.

The first generation of software product line methods emphasized a clear dichotomy between the activities of domain engineering and those of application engineering. The role of domain engineers

was to create core software assets “for reuse” and the role of application engineers was to use the core assets to create products “with reuse”. This was a logical extension of component-based software reuse, though in this case the components were designed for a particular application domain and with a particular software architecture in mind.

With application engineering, an individual product is created by configuring and composing the reusable *core assets*. Configuration is the instantiation of predefined variation *within* a core asset, either via automated mechanisms such as template instantiation and `#ifdefs`, or via manual techniques such as writing code to fill out function stubs. Composition is the development of “glue code” and application logic *around* the core assets. Application engineers are typically guided by a written *production plan* on how to perform the configuration and composition.

At first glance, all seems well. Products are created with much less effort due to reuse of the core assets. If the engineering effort for each product ended with its creation, then all would indeed be well. However, most software product lines must be maintained and must evolve over time, and this is where the problems with application engineering arise.

The first problem is that product-specific software in each product is just that – product-specific. It is one-of-a-kind software that often requires a team of engineers dedicated to the product to create it, understand it, maintain it, and evolve it. This takes us back to the problems of conventional software development approaches, such as clone-and-own, where there is a strong linear relationship between increasing the number of products in the product line portfolio and the increasing the number of engineers required to support the product line.

The second problem is that application engineering creates a unique and isolated context in which the core assets are reused. This makes it difficult to enhance, maintain or refactor the core assets in nontrivial ways. Any syntactic or semantic changes to the interfaces or architectural structure of the core assets have to be merged into the product-specific software in each and every one of the products in the product line. This can be very error prone and very costly. The effect of this high cost of evolution is to stifle innovation and evolution for the product line. Also, significant upfront effort is required to create immutable and pristine product line architectures and reusable assets so that evolution and maintenance is minimized.

The third problem is that the hard delineation between domain engineering and application engineering creates hard structural boundaries in the software and in the organization. Reusable abstractions that emerge during application engineering go unnoticed by the domain engineering team since it is outside of their purview. Neither domain engineers nor

application engineers can see valuable opportunities for refactoring that may exist across the boundary between core assets and product-specific software.

The fourth problem is that the organizational delineation between domain engineering teams and application engineering teams creates cultural dissonance and an “us-versus-them” tension within a development organization.

With the software mass customization methodology, the negative effects of manual application engineering are avoided by focusing almost entirely on domain engineering and fully automated production using software product line *configurators*.

2.2. Software Product Line Configurators

As shown in Figure 1, configurators take two types of inputs – core software assets and product models – in order to automatically create product instances. The core software assets can include requirements, architecture and design, source code, test cases, product documentation, and so forth. The product models are concise abstractions that characterize the different product instances in the product line, typically expressed in terms of a *feature model*[5].

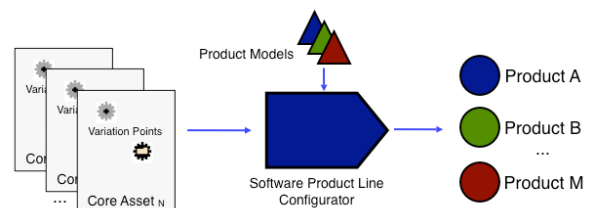


Figure 1. Software Product Line Configurator

The predominate development effort with configurators is domain engineering of the core assets. There is a relatively small effort that goes into the product models, which essentially replaces the manual application engineering effort.

At first glance, this appears similar to the use of abstraction in conventional source code compilers, software generators, or model-driven development compilers. However, configurators work more by composition and configuration of conventional assets rather than generation, transformation, or translation of one representation into another representation.

Software mass customization with configurators provides a simpler, more efficient, and more scalable solution than the DE/AE dichotomy.

- Since all software exists within a consolidated collection of core assets, everything is candidate for reuse. This is true even for product-specific assets that start out as unique to one product.
- Since all development is focused on core asset development, teams are organized around the assets.

This organizational structure looks similar to product-centric development and is the same for 2 products or 200 products, making it highly scalable.

- Evolution of the core assets and products is synonymous. For example, a change in the core assets can be followed by automated re-instantiation of all products to reflect that change. No manual merging and reintegration is required.

2.3. Developing the Intangible Product

With the software mass customization methodology, products only become tangible after the automated production step. Under this model, there are no longer tangible teams of application engineers organized around tangible products. And yet, the need remains to have organized and tangible product release plans and to deliver products according to tangible release schedules.

The key to managing product releases is in synchronizing the release of features in core assets with the product release schedules. In order to release a product at a given point in time, the core assets used in the composition and configuration of that product must meet or exceed the required functionality and the required quality for that product release.

Product managers in this scenario are not managing tangible teams of application engineers to meet their release schedules. They manage their schedules by coordinating and negotiating with architects and core asset developers to make sure that the core assets meet or exceed the required functionality and quality. From the perspective of the core asset teams, their release schedules are driven by the consolidated requirements and schedules from all of the products.

3. Minimally Invasive Transitions

Work Like a Surgeon, Not Like a Coroner

The methodology of *minimally invasive transitions* is distinguished by minimizing the cost, time and effort required for organizations to adopt software product line practice. The key is the minimal disruption of ongoing production schedules during the transition from conventional product-centric development practice. Minimally invasive transitions enable transitions in two orders of magnitude less effort than that experienced using earlier methods[2,3].

The term minimally invasive transitions is intended to invoke the analogy to medical procedures in which problems are surgically corrected with minimal negative side-effects to the patient. In both cases – medical and software – excessive disruption can be very detrimental to the patient.

The motivation behind minimally invasive transitions is to address one of the primary

impediments to widespread use of early generation software product line development methods – an imposing and often times prohibitive adoption barrier.

The long-term benefits of early generation software product line methods would suggest an easy business case to justify adopting a software product line approach. However, as part of the business case, resources for the huge upfront investment, estimated as equal to effort of developing 2 to 3 products from scratch, have to be allocated[1,6]. The problem is not so much the money as it is the human resources. Diverting that amount of the existing expert engineering resources from product development to software product line transition inevitably means that production schedules are disrupted. In a classic early generation software product line approach, Cummins diverted nine contiguous months of their overall development effort into a software product line transition effort[1]. For most product development companies, this level of disruption to ongoing production schedules cannot be tolerated.

In contrast, for minimally invasive transitions, the benefits of software product line methods can be achieved with a much smaller, non-disruptive transition investment and with much faster return on investment. Two primary techniques are employed for this methodology, (1) reusing as much as possible of the existing assets, processes, infrastructure, and organizational structures, and (2) incremental transitions in which small upfront investments offer immediate and incremental return on investment.

3.1. Minimize the Distance to a Better Product Line Practice

Most product development organizations in today's markets create a portfolio of products – a product line – rather than just a single product. Thus, for software development teams, the idea of creating software for a product line is not new, since they are most likely already doing that. What is new are some of the “true” software product line development methods, tools and techniques that make product line development easier and more efficient.

By nature, engineers prefer to re-engineer new solutions from a greenfield rather than brownfield, but that is often ruled out by the business case. Re-engineering a product line approach from scratch is too expensive and causes too much disruption to ongoing production schedules. What often makes the most sense is a more surgical approach that fixes the biggest problems while reusing as much as possible from the current approach.

For example, with software mass customization configurators, it is possible to reuse most of the legacy software assets, with just enough refactoring to allow for composition and configuration by the configurators.

Existing architecture and infrastructure is often sufficient with little or no modification. The initial feature model and initial variation points in the consolidated core assets do not need to be elegant, as long as they accurately instantiate product instances. The initial versions can be refined later[3].

In terms of product line scope, the shortest path to a live product line deployment is to take the minimalist approach. Initially the core assets should support the products that need to be deployed in the near term. Then a reactive style of scoping can be utilized to evolve the core assets and production line as business demands dictate[7].

3.2. Incremental Return on Incremental Investment

Early generation software product line methods suggested that a large upfront investment was required in order to gain the even larger benefits of software product line practice. However, with minimally invasive transition techniques, it is also possible to achieve the same results by making a series of smaller incremental investments. By staging an incremental transition, small incremental investments very quickly yield much larger incremental returns. These returns – in effort, time and money – can then be partially or fully reinvested to fuel the next incremental steps in the transition. Furthermore, the efficiency and effectiveness of the development organization constantly improves throughout the transition, meaning that development organizations do not need to take a hit in order to reap the benefits of software product line practice.

For example, Engenio achieved return on investment after an incremental investment of only 4 developer-months of effort. In contrast, the conventional wisdom from first generation software product line methods predicted that the upfront investment before getting any return for Engenio would be 900 to 1350 developer-months, or 200 to 300 times greater than that actually experienced[3].

There are, of course, many different facets within a software development organization to consider when making an incremental transition to software product lines. For example, Clements and Northrop characterize 29 key practice areas that may be impacted by a software product line approach[1]. Any or all of these might be considered in an incremental transition.

Engenio chose to incrementally address, in sequence, those facets that represented the primary inefficiencies and bottlenecks in their development organization. By eliminating the inefficiencies and bottlenecks in the most critical facet, the next most critical product line problem in the sequence was exposed and targeted for the next increment[3].

4. Bounded Combinatorics

As a practical limit, the number of possible products in your product line should be less than the number of atoms in the universe.

The methodology of bounded combinatorics focuses on constraining, eliminating, or avoiding the combinatoric complexity in software product lines. The combinatoric complexity, presented by the multitude of product line variations within core assets and feature models, imposes limitations on the degree of benefits that can be gained and the overall scalability of a product line approach. Bounded combinatorics overcome the limitations present in early generation product line methods, opening new frontiers in product line practice.

Some simple math illustrates the problem. In product lines with complex domains, companies have reported feature models with over 1000 feature variations[8]. Note, however, that the combinatorics of only 216 simple boolean features is comparable to the estimated number of atoms in the entire universe. Clearly the full combinatoric complexity of 1000 varying features is not necessary in any product line. Clearly there is great benefit to methods that bound the astronomical combinatorics and bring them in line the relatively small number of products needed from any product line.

The limitations imposed by combinatoric complexity in a product line are most prominent in the area of testing and quality assurance. Without highly constrained combinatorics, testing and validating all of the possible feature combinations is computationally and humanly intractable[9].

A combination of new and standard software engineering techniques are applied in innovative ways to form the bounded combinatorics methodology.

4.1. Modularity, Encapsulation and Aspects

The large, monolithic feature models characteristic of early generation product line approaches are like large global variables. Any feature can impact any core asset and any core asset may be impacted by any feature. This has all of the software engineering comprehension drawbacks as global variables in conventional programming languages, so the bounded combinatorics methodology utilizes the same modularity and encapsulation techniques that are utilized to eliminate the use of global variables in programming languages.

Modularity is applied to partition the feature model into smaller models. The criteria for partitioning is to localize each feature within the smallest scope that it needs to influence. For example, some features may only impact a single core asset component and should be encapsulated in a feature model partition for that

component. Some features may impact all of the core assets components within a single subsystem and therefore should be encapsulated in a feature model partition for that subsystem. Some features may be aspect-oriented and need to cut across multiple core assets, so these should be encapsulated into an aspect-oriented feature model partition and “imported” into the scope of the core asset components or subsystems that need them.

A new abstraction call a *feature profile* is used to express which feature combinations within a partition are valid for use in the configuration and composition of the core asset components, subsystems, and aspects, often reducing the combinatorics for a module from exponential and astronomical to the linear list of valid feature profiles[5].

4.2. Composition and Hierarchy

Another useful abstraction for bounding combinatorics in a software product line is the *composition*. A composition is a subsystem that is composed of core asset components and other compositions. The role of the composition is to two-fold: (1) to encapsulate a feature model partition for the subsystem represented by the composition (as described in the previous section), and (2) to encapsulate a list of *composition profiles* that express the subset of valid combinations of the child components and nested compositions[5]. Composition profiles reduce combinatorics to a similar degree as feature profiles at each level in the composition hierarchy[5].

Each composition can be treated as a smaller, independent product line, nested within the context of the larger product line, which has many powerful implications. The first is that the composition hierarchy is really a nested product line hierarchy. The second is that each nested product line can be used and reused in multiple different product lines, extending the notion *product populations*[10]. Third, decomposing a large product line into a collection of smaller product lines makes it easier to allocate smaller development teams to each of the smaller nested product lines. Fourth, smaller nested product lines offer another opportunity for making incremental, minimally invasive transitions into product line practice.

5. Conclusions

The first generation of software product line methods extended software engineering efficiency and effectiveness for a portfolio of products to unprecedented levels. The next generation of software product line methodology offers another big step forward, opening a new realm of strategic and competitive advantages over conventional

development and first generation product line methods. Software mass customization methodology and configurator technology eliminates one of the biggest inefficiencies of first generation approaches – manual application engineering. The minimally invasive transition methodology eliminates one of the long-standing impediments to the widespread use of first generation software product line approaches – the adoption barrier. The bounded combinatorics methodology pushes the complexity barrier many orders of magnitude beyond what was possible in the first generation methods.

6. References

- [1] Paul Clements and Linda Northrop. 2001. *Software Product Lines: Practice and Patterns*, AddisonWesley, Reading, MA.
- [2] Ross Buhrdorf, Dale Churchett, Charles Krueger. *Salion's Experience with a Reactive Software Product Line Approach*. 5th International Workshop on Product Family Engineering. Nov 2003. Siena, Italy. Springer-Verlag LNCS 3014, p 315.
- [3] William A. Hetrick, Charles W. Krueger and Joseph G. Moore, *Incremental Return on Incremental Investment: Engenio's Transition to Software Product Line Practice*, October 2006, OOPSLA 2006, Portland, Oregon, ACM.
- [4] Krueger, C. *Easing the Transition to Software Mass Customization*. Proceedings of the 4th International Workshop on Product Family Engineering. October 2001. Bilbao, Spain. Springer-Verlag, New York, NY.
- [5] See *BigLever Software Gears* data sheet at http://www.biglever.com/extras/Gears_data_sheet.pdf.
- [6] Davis Weiss and Chi Tau Robert Lai. 1999. *Software Product-line Engineering*. Addison-Wesley, Reading, MA.
- [7] Clements, P. and Krueger, C., *Being Proactive Pays Off / Eliminating the Adoption Barrier*. IEEE Software, Special Issue of Software Product Lines. July/August 2002, pages 28-31.
- [8] Mirjam Steger, et.al., *Introducing PLA at Bosch Gasoline Systems*, proceeding of the Third International Conference, SPLC 2004, Boston, MA, Aug/Sep 2004, Springer-Verlag LNCS 3154, p 34.
- [9] Proceedings of the 2005 Software Product Line Testing Workshop, http://www.biglever.com/split2005/Presentations/SPLiT2005_Proceedings.pdf.
- [10] Rob van Ommering, *Widening the Scope of Software Product Lines – from Variation to Composition*, proceeding of the Software Product Lines 2nd International Conference, SPLC 2, San Diego, CA, Aug 2002, Springer-Verlag LNCS 2379, p 328.