

# NEW METHODS IN SOFTWARE PRODUCT LINE PRACTICE

Examining the benefits of next-generation SPL methods.

**T**ools and techniques for software development tend to focus on developing individual products. However, most development organizations must create a product line—a portfolio of closely related products with variations in features and functions—rather than a single product. This mismatch has led to the emergence of software development methods, tools and techniques, focused specifically on the challenges of software product line (SPL) development.

Early SPL case studies, at the genesis of the field, revealed some of the best software engineering improvement metrics seen in four decades [3]. However, they also demonstrated significant barriers to adoption and challenges in ongoing practice [2].

Some of the latest generation of SPL success stories use methods that dramatically reduce these barriers and challenges, enabling the SPL development approach to move more easily into mainstream practice. In this article, three of the new methods behind the latest generation of SPL success stories are explored. The topics discussed here are based on first-hand experience with industry projects 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 [1, 5].

### SOFTWARE MASS CUSTOMIZATION

*Application Engineering Considered Harmful.* 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 to create different products. 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 characterization of the product’s desired feature profile [6]. Compared to the first-generation SPL development methods, which relied heavily on manual development effort and processes, the high degree of automation associated with software mass customization makes this method easier to adopt and sustain.

The first-generation methods emphasized a clear dichotomy between the activities of domain engineering and those of application engineering. The role of domain engineers was to create software core assets “for reuse” and the role of application engineers was to use these reusable core assets to create products “with reuse.” This was a logical extension of component-based software reuse.

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 SPLs must be maintained and must evolve over time, and this is where the problems with application engineering arise.

The first problem with application engineering is that developers manually create product-specific configuration and “glue code” in each product. This is

one-of-a-kind software that often requires a team of engineers dedicated to the product. Increasing the number of products in a product line requires a linear increase in the number of developers.

The second problem is that each product has a unique and isolated context in which the core assets are reused. This makes it difficult to enhance the core assets in nontrivial ways. Changes must be merged back into the product-specific software in each and every product in the product line.

The third problem is that the organizational delineation between domain engineering teams and application engineering teams creates an “us-versus-them” culture. When development problems arise, which side is causing the problems? Which side is responsible for fixing the problems? This organizational dissonance decreases development efficiency and effectiveness.

With the software mass customization methodology, the negative effects of application engineering are avoided using SPL configurators in place of application engineers. SPL configurators

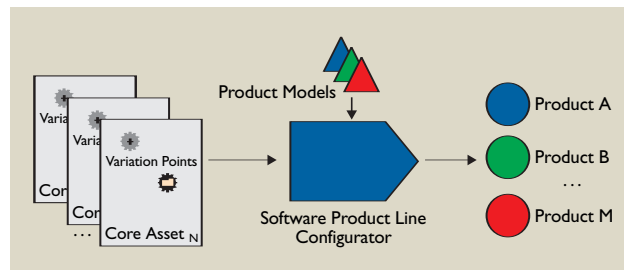


Figure 1. Software product line configurator.

enable SPL development via automated product instantiation rather than manual application engineering. As shown in Figure 1, configurators take two types of inputs—core assets and product models—in order to automatically create product instances. The core assets are the common and varying software assets for the product line, such as requirements, architecture and design, source code, test cases, and product documentation. The product models are concise abstractions that characterize the different product instances in the product line, expressed in terms of a feature model [4].

With SPL configurators, all developed software exists within the consolidated collection of core assets, making everything a candidate for reuse and refactoring. Since all development is focused on core asset development, teams are organized around the different assets. This organizational structure eliminates the need to add a development team each time a new product is added. Evolution of the core assets and products is coincident: any change to the core assets can be followed by automated re-instantiation of all products, without manual merging.

### MINIMALLY INVASIVE TRANSITIONS

*Work Like a Surgeon, Not Like a Coroner.* The term minimally invasive transitions is intended to invoke the

analogy to medical procedures in which problems are surgically corrected with minimal degrees of disruption and negative side effects to the patient. The SPL development methodology of minimally invasive transitions is distinguished by the minimal disruption of ongoing production schedules during the transition from conventional product-centric development practice.

Minimally invasive transitions take advantage of existing software assets and rely on incremental adoption strategies to enable transitions in two orders of magnitude less effort than that experienced using earlier methods that relied on upfront, top-down reengineering of significant portions of software assets, processes, and organizational structures [1, 5].

The motivation behind minimally invasive transitions is to address one of the primary impediments to widespread use of early generation SPL development methods—an imposing and often prohibitive adoption barrier. Figure 2 extends the canonical SPL economic graph introduced by Weiss [10]. It illustrates the cost and effort required to build and maintain a SPL under three different product line development methodologies: conventional product-centric development, early generation SPL methodologies, and minimally invasive transition methods. The graph shows the rate of increasing cost and effort as more products are added to the product line portfolio (the slope of the lines), the upfront investment required to adopt the product line practice (the vertical axis intercept), and the break-even, return on investment (ROI) points relative to conventional product-centric approaches.

Comparing the product-centric to the early-generation SPL methods would suggest an easy business case to justify adopting a SPL approach. However, as part of the business case, resources for the upfront investment must be allocated. Diverting the existing expert engineering resources from product development to SPL transition inevitably means that production schedules are disrupted. In a classic early generation SPL approach, Cummins diverted nine contiguous months of their overall development effort into a SPL transition effort [3]. For most product development companies, this level of disruption to ongoing production schedules cannot be tolerated.

In contrast, the graph for minimally invasive transitions illustrates that the benefits of SPL methods can

be achieved with a much smaller, non-disruptive transition investment and with much faster ROI [1, 5]. Two primary techniques are employed for this methodology.

The first method is carefully assessing how to reuse as much as possible of an organization's existing assets, processes, infrastructure, and organizational structures. For example, with 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.

The second technique is finding an incremental transition approach such that a small upfront investment creates immediate and incremental ROI. The excess returns from the initial incremental step can be reinvested to fuel the next incremental step—then repeated as needed. The efficiency and effectiveness of the development organization constantly improves throughout the transition, meaning that development organizations do not need to disrupt ongoing production.

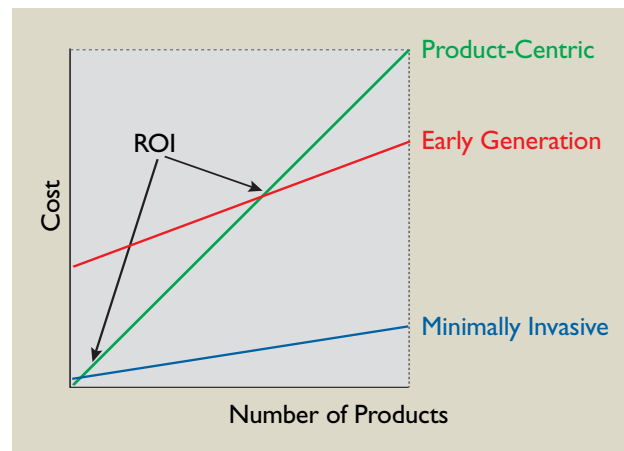


Figure 2. Upfront Investment and ROI.

### 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 SPLs. 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. New methods for bounded combinatorics reduce combinatorics from exponential and astronomical to linear and humanly tractable.

Some simple math illustrates the problem. In product lines with complex domains, companies have reported feature models with over 1,000 feature variations [9]. Note, however, that the combinatorics of only 216 simple Boolean features is comparable to the estimated number of atoms in the entire universe. With only 33 Boolean features, the combinatorics can represent one product for every human on the planet.

The limitations imposed by combinatoric complexity in a product line are most prominent in the area of testing and quality assurance. Without highly con-

strained combinatorics, testing and validating all of the possible feature combinations is computationally and humanly intractable [8].

Two techniques are applied to form the bounded combinatorics methodology. The first is a combination of modularity, encapsulation, and aspects. The second is a combination of composition and hierarchy.

*Modularity, encapsulation, and aspects* are 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 with that component. Some features may impact all of the core assets components within a single subsystem and therefore should be encapsulated with 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.

This partitioning allows us to apply encapsulation to hide much of the combinatoric complexity associated with a feature model partition. A new abstraction called a feature profile expresses which feature combinations within a partition are valid for use in the configuration and composition of the core asset components, subsystems, and aspects (see [www.biglever.com/extra/Gears\\_data\\_sheet.pdf](http://www.biglever.com/extra/Gears_data_sheet.pdf)).

For example, imagine a feature model for a core asset component that has five independent variant features, each with four possible values. The full combinatorics for this feature model says that there are  $4^5$ , or 1,024, different ways of instantiating the component. However, we may only utilize eight of these combinations for our current product line. By declaring the eight feature profiles, we are specifying that only these eight possible configurations are valid for use. This encapsulation of feature combinations has reduced the combinatoric complexity by a factor of 128 for this one core asset.

Composition and hierarchy provide another useful abstraction for bounding combinatorics. A new abstraction called a composition denotes a subsystem that is composed of core asset components and other compositions. In other words, a SPL can be represented as a tree of compositions and components, where the components are the leaves of the tree and the compositions are the internal nodes of the tree.

The role of the composition is twofold: to encapsulate a feature model partition for the subsystem represented by the composition (as described earlier), and to encapsulate a list of composition profiles that express the subset of valid combinations of the child

components and nested compositions (see [www.biglever.com/extra/Gears\\_data\\_sheet.pdf](http://www.biglever.com/extra/Gears_data_sheet.pdf)). Similar to feature profiles, the composition profiles can reduce the combinatorics by multiple orders of magnitude at each node in a composition hierarchy.

## CONCLUSION

The first generation of SPL methods extended software engineering efficiency and effectiveness for a portfolio of products to unprecedented levels. After pushing these methods to their limits and gaining better understanding what those limits are, the next generation of SPL methods offers another big step forward. Three important new methodologies described in this article are software mass customization, which eliminates labor-intensive application engineering, minimally invasive transitions, which eliminates the adoption barrier, and bounded combinatorics, which extends the scalability of product line portfolios. For more coverage of this topic, see [7].

In combination, these new methods offer tactical benefits that are large enough to open a new realm of strategic and competitive advantages over conventional software development and first-generation SPL development methods. **C**

## REFERENCES

1. Buhrdorf, R., Churchett, D., and Krueger, C. Salion's experience with a reactive software product line approach. In *Proceedings of the 5th International Workshop on Product Family Engineering* (Siena, Italy, Nov. 2003), Springer-Verlag LNCS 3014, 315.
2. Clements, P. and Krueger, C. Being proactive pays off: Eliminating the adoption barrier. *IEEE Software Special Issue of Software Product Lines*. (July/August 2002), 28–31.
3. Clements, P. and Northrop, L. *Software Product Lines: Practice and Patterns*. Addison Wesley, Reading, MA, 2001.
4. Czarnecki, K. and Eisenecker, U. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, MA, 2000.
5. Hetrick, W.A., Krueger, C.W., and Moore, J.G. Incremental return on incremental investment: Engenio's transition to software product line practice. In *Proceedings of OOPSLA 2006*, (Portland, OR., Oct. 2006)
6. Krueger, C. Easing the transition to software mass customization. In *Proceedings of the 4th International Workshop on Product Family Engineering* (Bilbao, Spain, Oct. 2001), Springer-Verlag, New York, NY.
7. New methods behind a new generation of software product line successes. Technical Report 200602261, BigLever Software; [www.biglever.com/forms/newmethods.html](http://www.biglever.com/forms/newmethods.html).
8. *Proceedings of the 2005 Software Product Line Testing Workshop*; [www.biglever.com/split2005/Presentations/SPLIT2005\\_Proceedings.pdf](http://www.biglever.com/split2005/Presentations/SPLIT2005_Proceedings.pdf).
9. Steger, M. et al. Introducing PLA at Bosch Gasoline Systems. In *Proceedings of the Third International Conference, (SPLC 2004)* (Boston, MA, Aug./Sept. 2004), Springer-Verlag LNCS 3154, 34.
10. Weiss, D. and Lai, R. *Software Product Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, Reading, MA, 1999, 73–74.

---

**CHARLES W. KRUEGER** ([ckrueger@biglever.com](mailto:ckrueger@biglever.com)) is the chief executive and founder of BigLever Software, in Austin, TX.

---