



Contact Information:

info@biglever.com
www.biglever.com
512-426-2227

New Methods Behind a New Generation of Software Product Line Successes

Report #200602261

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 unintentional occurrences 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. The dichotomy of domain engineering (DE) and application engineering (AE) introduces dissonance and inefficiency in software development. The *software mass customization* methodology utilizes abstraction and automation to eliminate application engineering and thereby the negative effects of the DE/AE dichotomy.
- Minimally Invasive Transitions. The cost, time and effort required for organizations to adopt early generation software product line methods is disruptive and prohibitive. *Minimally invasive transition* methods enable rapid adoption of product line practice with low disruption, low cost, and rapid return on investment.
- Bounded Combinatorics. Combinatorics in feature models, requirements, architecture, implementation variation points, and test coverage can easily exceed the number of atoms in the universe, which leads to complexity, errors, and poor testability. New methods for *bounded combinatorics* utilize abstraction, modularity, encapsulation, composition, hierarchy, and constraints to reduce combinatorics from geometric, exponential and astronomical to simple, linear and tractable.

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. A key characteristic of this approach is the synchronous evolution of all products in the product line in conjunction with the maintenance and evolution of the reusable software assets. Software mass customization utilizes a combination of formal abstraction and automation to enable more effective reuse and higher overall software product line development efficiency compared to earlier methods that relied heavily on manual application engineering.

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[2].

The motivations behind the software mass customization methodology are both economic and pragmatic. Compared to the first generation software product line development methods, software mass customization is easier to adopt, has greater similarity to conventional software development, is less disruptive to organizational structures and processes, and offers greater and faster returns on investment. Furthermore, it reduces product creation and evolution to a commodity, so that adding new products to a product line can be trivial.

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 in a way that is consistent with the product line architecture.

At first glance, all seems well. Products are created with much less effort due to reuse of core assets, including the production plan and product line architecture. 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. These problems can best be characterized by the dichotomy that exists between core assets and product-specific software, as the entire product line evolves over time.

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.

Because the product-specific software exists in the context of each individual product and its associated application engineering team, this software cannot be easily reused. It is likely that a significant amount of similar software will be independently developed across the different products, but the potential for reuse of emerging abstractions will go unnoticed because of the isolated application engineering teams and the independent, diverging product software branches.

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, both in terms of time and effort. 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 political tension within a development community. This organizational dichotomy within the development community is very different from conventional organizational structures, so it can be difficult to initially partition the team. The dichotomy creates a new cultural “us-versus-them” tension. All of the technical software development problems described earlier were due to the challenges at the domain engineering and application engineering interface, which by human nature leads to an acrimonious and unproductive relationship when problems arise. Which side is causing problems? Which side is responsible for fixing problems? Organizational dissonance, of course, decreases the overall efficiency and effectiveness.

A common question in early generation software product line approaches was what percentage of effort should be spent on domain engineering and what percentage of effort should be spent on application engineering. Figure 1 illustrates this question. By plotting the accumulated overall cost of developing some number of products in a product line (six products in this example), the green “slider” on the X-axis shows the Y-axis cost for any given ratio of DE to AE. Given the cost plots for *Product #1* through *Product #6* in this figure, it is most advantageous to shift the balance as close as possible to 100% domain engineering. But are the cost plots in Figure 1 realistic and is it possible for a software product line methodology to be predominately focused on domain engineering?

The software mass customization methodology is an affirmative answer to both of these questions. The negative effects of the DE/AE dichotomy and those of manual application engineering are avoided by shifting the balance away manual application engineering and focusing entirely on domain engineering and fully automated production using technology called software product line *configurators*.

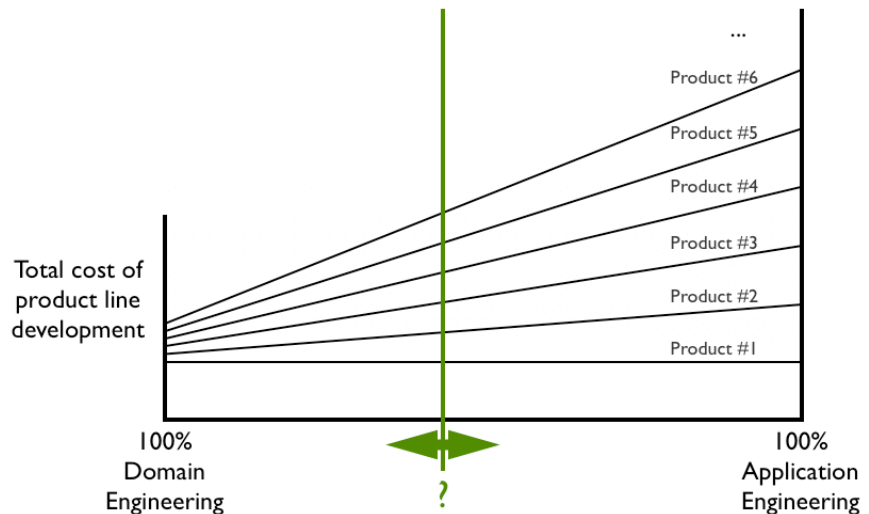


Figure 1. Optimal Balance of Domain Engineering and Application Engineering?

2.2. Software Product Line Configurators

Software product line configurators are a class of technology that enables software mass customization – software product line development based on automated product instantiation rather than manual application engineering. As shown in Figure 2, configurators take two types of input – core software assets and product models – in order to automatically create product instances. The core software 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* – the feature profile of the products. BigLever Software *Gears* is an example of a commercial software product line configurator[3].

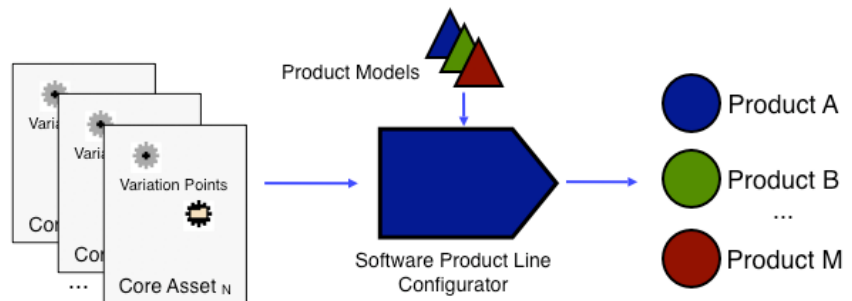


Figure 2. Software Product Line Configurator

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 that dominates in the DE/AE dichotomy.

With configurators, the product models are a level of abstraction over the core assets. 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 of software into another representation.

Configurators illustrate the analogy between the practice of mass customization in manufacturing and the practice of software mass customization. In manufacturing, parts that are specially engineered with variations are composed and configured by an automated factory, based on a feature profile for the product instance under manufacture.

Contrasting to the four problems outline earlier in the DE/AE dichotomy, software mass customization with configurators provides a simpler, more efficient, and more scalable solution.

- 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 the core assets are all within a single context, it is much easier to observe emerging abstractions and to refactor the assets, including requirements, architecture, source code, and test cases.
- Since all development is focused on core asset development, teams are organized around the different assets. This organizational structure looks very similar with 2 products or with 200 products, making for a scalable solution. This eliminates the strong linear relationship between the number of products and staff size that we observed with the DE/AE dichotomy.
- 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.
- Organizational structures are similar to conventional software development, where teams are organized around subsystems. With software product line development, developers need to introduce and manage the variation within core assets, but this is also familiar to any developer that has used primitive variation management techniques such as `#ifdefs`.

With software mass customization, there is very little overhead to adding a new product to a product line. New products are commodities rather than the subject of a major long-term development and maintenance commitment. If the existing core assets are sufficient to support a new product, then all that is needed is a new product model. If the existing core assets are not sufficient, then the core assets are extended with first class development on the core assets, not as one-off glue code in a product-specific code branch.

Although the software mass customization model is a simpler and more familiar methodology than the DE/AE dichotomy, there are two new techniques that need to be addressed: *developing the intangible product* and *variation management in space and time*.

2.3. Developing the Intangible Product

With the software mass customization methodology, where automated configurators replace manual application engineering, 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. How do you manage the domain engineering of the core assets so that release schedules for individual products can be met?

The key to managing product releases is synchronizing the release of features and capabilities 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.

Figure 3 illustrates an example of the synchronized release of core assets and products to meet different product release requirements. Time is represented across the top horizontal axis, increasing from left to right. with each baseline (Baseline 1, Baseline 2, ...), occurring on monthly intervals. In between the baselines are weekly and daily

synchronization points – such as daily automated builds and weekly integration tests – to manage the evolution and convergence.

Across the bottom of the figure are the product release schedules for each of the products, A through M. Over time, each product is scheduled to cycle through quality validation levels, which in this example are alpha (red triangles) for initial validation, beta (yellow diamonds) for preliminary release, and general availability (GA, green pentagons) for final release. Note that these product release points are synchronized on the major release baselines (Baseline 1, Baseline 2, ...).

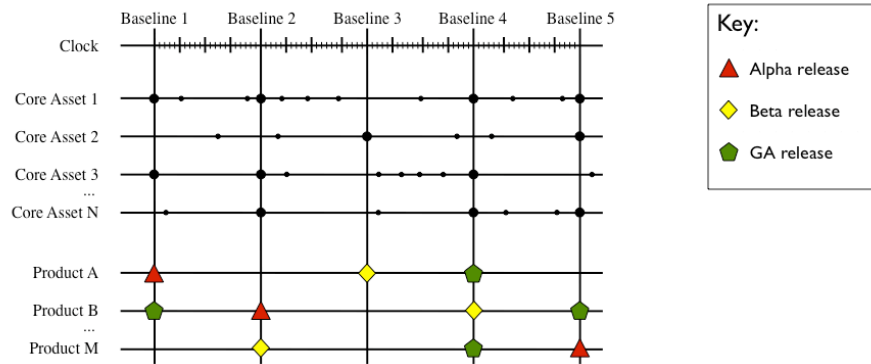


Figure 3. Synchronizing the Release Clocks for Core Assets and Products

Product managers in this scenario are not managing tangible teams of application engineers to meet their release schedules. They must manage schedules by coordinating and negotiating with product line architects and core asset developers to make sure that the core assets meet or exceed the required functionality and quality on specific baselines.

From the perspective of the core asset teams (Core Asset 1, Core Asset 2, ...), their release schedules are driven by the consolidated requirements and schedules from all of the products. For the software mass customization methodology to work efficiently and effectively, it is essential to have effective architectural planning, coordination and synchronization at this level. The product managers and architects represent the interest of the product release schedules in order to provide the core asset development managers with well defined and realistic core asset release requirements.

A “release” in this methodology is a baseline of the core assets. As shown in Figure 3, it is not essential that all products be supported on each baseline, or that all products be released at the same quality state (alpha, beta, general availability). However, there is an advantage to synchronize the release of multiple products on one baseline. You can amortize the release overhead, such as testing and documentation, by clustering multiple product releases on a single baseline.

2.4. Variation Management in Time and Space

Core assets and product models evolve over time under the software mass customization methodology, just like conventional software development. Conventional configuration management and component baseline management can be used to manage this *variation in time*.

Different from conventional one-of-a-kind software development, variations within the domain space of the product line must also be managed. At any fixed point in time, such as a core asset baseline, it is the core asset variation points, product models and configurators manage this *variation in space*.

Figure 4 illustrates the distinction between variation in time and variation in space. In the top portion of the figure, core assets and product models are evolving over time, with the baseline representing stable and compatible releases of the core assets. In the bottom of the figure, the configurator takes core assets and product models from a

given baseline and instantiates the different products within the domain space of the product line.

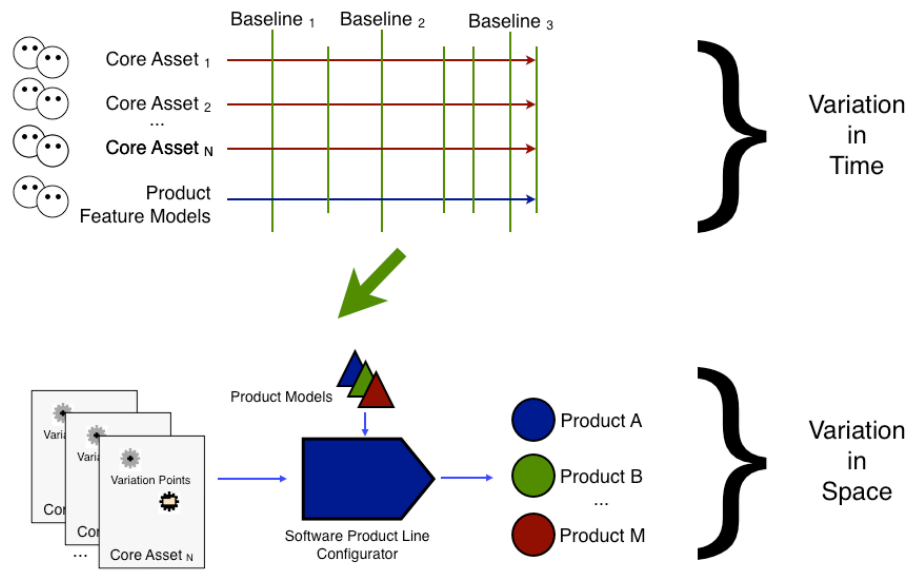


Figure 4. Software Variation in Time and Space

This clean separation of concerns between variation in time and variation in space manifest in the software mass customization methodology assures that software product line development is familiar and requires the least amount of disruption when transitioning to a software product line approach. More details on this separation of concerns can be found in [4].

2.5. Impact of the Software Mass Customization Methodology

Software mass customization offers one of the most efficient forms of software reuse possible, according to the taxonomy in [12]. The *cognitive distance* between a product concept and the specification of that product in a feature model is very small, typically measured in developer-minutes or developer-hours. Configurator technology fully automates product instantiation from the feature model specification for a product. Furthermore, configurators can reuse legacy assets and core assets developed using conventional technology and techniques, so creating a the core assets for a product line utilizes the conventional skills of existing development teams.

The impact of this methodology is that products in a product line become commodities. The cost of adding and maintaining a new product in a product line portfolio is almost trivial, making it possible to scale portfolios to include orders of magnitude more products than is possible under early generation methods that rely on manual application engineering and the labor intensive AE/DE dichotomy.

3. Minimally Invasive Transitions

Work Like a Surgeon, Not Like a Coroner

The software product line development methodology of *minimally invasive transitions* is distinguished by its focus minimizing the cost, time and effort required for organizations to adopt software product line practice. A key characteristic of this methodology is 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 re-

engineering of significant portions of software assets, processes, and organizational structures[5,6].

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. In both cases – medical and software – excessive and extraneous 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. Figure 5 illustrates the cost and effort required to build and maintain a software product line under three different product line development methodologies: conventional product-centric development, early generation software product line 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 up-front investment required to adopt the product line practice (the vertical axis intercept), and the return-on-investment (ROI) points relative to conventional product-centric approaches.

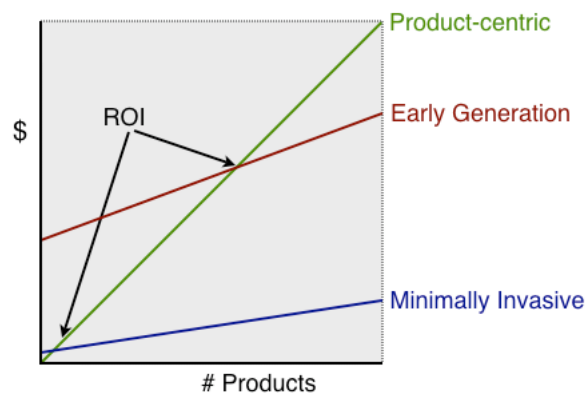


Figure 5. Upfront Investment and ROI

Comparing the product-centric to the 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 upfront investment have to be allocated. The problem is not so much the money as it is the human resources. Diverting 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. It would kill many companies before return on investment was ever achieved.

In contrast, the graph for minimally invasive transitions illustrates that the benefits of software product line methods can be achieved with a much smaller, non-disruptive transition investment and with much faster ROI. Two primary techniques are employed for this methodology. The first is carefully assessing how to reuse as much as possible of an organization's existing assets, processes, infrastructure, and organizational structures. The second is finding an incremental transition approach that will enable a small upfront investment that will offer immediate and incremental ROI. The excess returns from the initial incremental step can be reinvested to fuel the next incremental step – then repeat as needed.

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. Although they may already have innovative solutions in place, organizations considering a transition to software product line practice have often reached the practical limit in some areas of their existing practice. They are looking to new software product development methods that provide better solutions in those areas.

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. In other words, find the shortest path and best ROI that achieves the required improvements in the least disruptive way. Referring back to our medical analogy, this minimally invasive approach is for the benefit of the patient rather than the surgeon.

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. In a stable software product line practice, feature models and variation points are typically undergoing continuous evolution and refactoring, so the initial version of these can be refined later.

To minimize disruption in organizational structures, follow the advice from Section 2 and don't introduce new and unfamiliar organizational structures such as the dichotomy of domain engineering and application engineering groups.

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. It's Just Like Single-system Development Except...

Early generation software product line case studies made software product line development look very different from convention single-system software development. New generation case studies have shown that there really doesn't need to be that much difference. For organizations using minimally invasive transitions, the before and after picture look very similar.

It is useful to view software product line development as being just like single-system development except for a few distinct differences. For example, referring back to Figure 2 on the software mass customization methodology, the core assets can be viewed as the “single system” that is created and maintained by the development organization. The core assets evolve in a familiar way under configuration management, as illustrated in Figure 4. The software product line configurator is analogous to a requirement management tool that instantiates a set of requirements, or a source code compiler. Even the fact that multiple products can be automatically produced from the core assets is not unfamiliar to a software developer that has used `#ifdefs`, generics, or templates.

Engineers that we speak to from the new generation success stories reinforce this point of view. They tell us that they don't see much difference in their job after making the transition to software product line practice. As they develop their core assets,

they now have to deal with variation points and multiple instantiations of the core asset, but the pressures to deliver on schedule and the output of their effort appears to be very familiar.

In fact, because of this perception of “business as usual”, the core asset developers are often surprised when they see the global improvements in productivity, time-to-market and quality following their transition to product line practice. Thus it is important for management to provide good feedback about the improvement metrics so that developers can get a strong sense of accomplishment for their new software product line development efforts.

The development role that often perceives the biggest difference between conventional methods and the new software product line methods is the project manager. Previously they probably managed product-specific development teams, requirements and schedules. Now they are managing either core asset development teams or product delivery schedules for intangible products. Referring back to Figure 3, the separation and coordination of product delivery schedules and core asset development is relatively unique to software product line development.

3.3. Regression Red Flags

Because the before and after picture can look quite similar with minimally invasive transitions, it can be difficult to remain true to the new software product line principles and not slip into old habits. How can you tell when you inadvertently regress into the familiar old ways of conventional engineering?

The best indicator is when you detect any focus on tangible products rather than intangible products. It is a red warning flag if developers are focused on specific products rather than features and capabilities in the core assets. Examples of regression red flags include:

- References to products in core asset source code
- Product names in core asset file names or directory names
- Product-specific branches in configuration management
- Product development teams working across multiple core assets

3.4. 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 the investment for Engenio would be 900 to 1350 developer-months, or 200 to 300 times greater than that actually experienced[6].

There are, of course, many different facets to consider within a software development organization 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 elimi-

nating 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[6].

3.5. Impact of Minimally Invasive Transition Methodology

The minimally invasive transition methodology eliminates the highly disruptive and investment-prohibitive adoption barrier exhibited by early generation methods, thereby removing one of the long-standing impediments to the widespread use of software product line approaches. Techniques for reusing as much as possible and changing as little as possible when an organization transitions to product line practice imply that minimal rework is required and that developers and managers stay in a familiar environment. Techniques for incremental return on incremental investment imply that a chain reaction can be started with just a little upfront investment in time and effort, resulting in ongoing returns that dominate the ongoing investments. Minimally invasive transitions are possible with two orders of magnitude less upfront investment and nearly immediate return on investment.

4. Bounded Combinatorics

As a practical limitation, 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 10^{65} number of atoms in the entire universe. With only 33 boolean features, the combinatorics is comparable to the human population of our planet. 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, the testing and validation 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. Abstraction, modularity, scoping, aspects, composition and hierarchy are applied to feature modeling, product line architectures, and core assets.

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 many 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.

By partitioning the feature model, we limit the impact of each feature to the smallest scope that it needs to impact: subsystem, component, or aspect. Furthermore, this partitioning allows us to apply encapsulation to further hide much of the combinatoric complexity inside of a feature model partition. For example, in BigLever Software *Gears*, we added a new abstraction call a *feature profile* 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[3].

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 1024, different ways of instantiating the core asset component. However, we may only need 8 of these for our current product line. By declaring the 8 feature profiles for the feature model partition, we are specifying that only these 8 possible configurations are “exported” and valid for use. This means that the remaining 1016 possible feature combinations are not exposed to the outside and therefore do not need to be developed, tested or validated for use. This encapsulation of feature combinations has reduced the combinatoric complexity by a factor of 128 for this one core asset.

This example is typical of real-world scenarios in product line practice. The geometric and exponential combinatorics present in feature model partitions can be drastically bounded to a linear list of “interesting” feature profiles.

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. In other words, a software product line 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.

Figure 6 illustrates the composition hierarchy in BigLever Software *Gears*. *Module 1* through *Module 4* are the core asset components the leaves of the hierarchy. *Mixin 1* and *Mixin 2* encapsulate the cross-cutting aspect-oriented concerns. *Matrix 1* through *Matrix 3* are the compositions that represent the subsystems in the software product line.

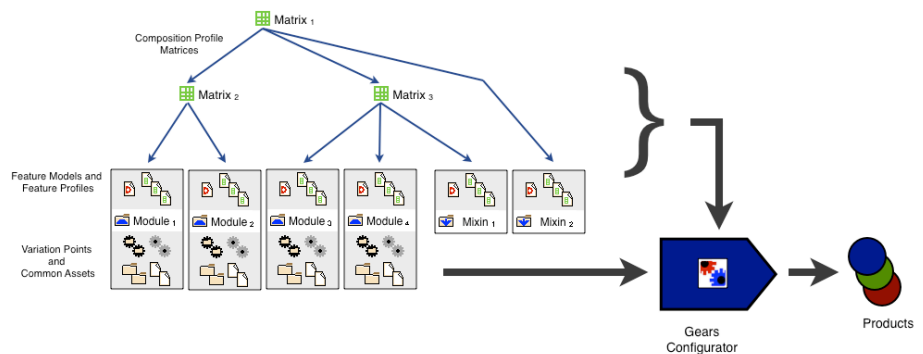


Figure 6. Modularity, Encapsulation, Aspects, Composition and Hierarchy

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 sec-

tion), and (2) to encapsulate a list of *composition profiles* that express the subset of valid combinations of the child components and nested compositions[3].

For example, imagine a subsystem composition with four children – two components and two nested compositions – where the two components each declare 8 valid feature profiles and the two nested compositions each declare 8 valid composition profiles. The full combinatorics for the composition of these children says that there are 8^4 , or 4096, possible compositions. However, most of these compositions are uninteresting and we may only need 16 for our current product line. By declaring 16 *composition profiles* for this composition, we are saying that we don't care about the other 4080 possible compositions, so we don't need to develop, test, or validate those compositions. We have reduced the combinatorics by a factor of 256 at this level in the composition hierarchy.

The combined use of feature model partitioning, feature profiles and composition profiles makes it possible to take the astronomical geometric and exponential feature combinatorics present in a monolith feature model and reduce it to a manageable linear number of configurations and compositions at each level in the composition hierarchy, including the root level that represents a complete product line.

The bounded combinatorics methodology adds two new abstraction layers over the early generation methods. In those early methods, the feature model served as an abstraction layer over the low-level variation points in the core assets. With bounded combinatorics, the feature profile adds an additional layer of abstraction over the feature model, describing valid configurations of core asset components. The composition profile adds a final layer of abstraction over the feature profiles to describe valid compositions of feature profiles and other composition profiles.

One final note of interest with compositions is that each composition can be treated as a smaller, independent product line, nested within the context of the larger product line. This view 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 of reusing common components in *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, the smaller nested product lines offers another opportunity for making incremental, minimally invasive transitions into product line practice.

4.3. Controlling the Product Line Scope

A common tendency when engineers are creating reusable software is to over generalize. It seem obvious that when you make a piece of software reusable, you should make it as generally reusable as possible. However, with over generality comes unnecessary cost, effort, and combinatoric complexity.

With software product line methods, it is important to pay special attention to this detrimental tendency since the entire product line is built out of reusable core assets. Early generation software product line methods sometimes prescribed generalizing core assets to satisfy predicted products on a 5-year horizon. However, Salion and BigLever Software illustrated that it can be more effective to be more agile and to generalize the core assets and product line architecture on a much shorter horizon – in their case between 3 to 6 months[5].

The distinction between very long range scoping and very short range scoping has been termed *proactive* versus *reactive* product line scoping[7]. Proactive is akin to the waterfall methods in conventional software development and reactive is akin to agile methods. This is really a spectrum rather than a boolean choice in software product lines. The choice on this spectrum is driven by business cases, the product marketing group, and market conditions.

Mistakes on either end of the spectrum – either being too proactive or too reactive – can be very expensive. You can make architecture mistakes that are difficult to refactor if you are too short-sighted. You may invest in a lot of throwaway develop-

ment effort if are over confident in your crystal ball such that your long-term predictions fail. In general, you will want to be more proactive with analysis and architecture, while being more reactive with the development of core assets.

Software mass customization and minimally invasive transition methods both shift the balance towards the reactive end of the spectrum. This lowers the overall combinatoric complexity of product line development.

4.4. Controlling Entropy

As with conventional software, product line assets can fall victim to *software entropy*, the gradual onset of disarray, degradation of structure, and loss of clarity that occurs in software as it is maintained and evolves over time. One of the most unanticipated and celebrated benefits of the software mass customization methodology is that it is not only possible to control entropy, but also straightforward to reverse it.

In early generation software product line methods, the multiple branches of product development with application engineering and AE/DE dichotomy created a situation that was highly susceptible to entropy and divergence in the code base. In contrast, the singular development focus on the consolidated collection of core assets with software mass customization is ideally structured for controlling entropy. It is straightforward to monitor reuse ratios and catch degradations before they take hold. It is easy for architects and developers to identify and refactor emerging abstractions within and among the explicit variation points in the core assets.

Applying the refactoring methods of agile development to their core assets, Salion watched the total lines of source code in their core asset base shrink by 30%, even as the number of products and the number of features in their product line increased by 400%. This innovative work is described in [11].

4.5. Impact of Bounded Combinatorics Methodology

Combinatoric complexity in early generation software product line methods created hard limits on the scalability and testability of a product line. Most commercial product lines have feature model, architecture, and variation point combinatorics that vastly exceed the number of atoms in the universe. Bounded combinatorics methods utilize a combination of abstractions and modularity – such a feature model partitions, feature profiles, hierarchical product line compositions, and composition profiles – to reduce these combinatorics by tens or even hundreds of orders of magnitude. This in turn extends the limits of what is possible in the scalability and testability of a product line by orders of magnitude.

5. Conclusions

The first generation of software product line methods extended software engineering efficiency and effectiveness for a portfolio of products to unprecedented levels. After pushing these methods to their limits and better understanding what those limits were, the next generation of software product line methodologies offer another big step forward. Three important new methodologies described in this paper are software mass customization, minimally invasive transitions, and bounded combinatorics.

Software mass customization capitalizes on configurator technology to automate the configuration and composition of core assets into products. This methodology allows us to eliminate one of the biggest inefficiencies of the first generation approaches – manual application engineering. The benefit of this methodology is that the cost of adding and maintaining new products to a portfolio reaches the commodity level. Product line portfolios can be scaled to much larger levels than previously possible. The entire product line can evolve with a comparable level of effort as evolving an individual product. Commercial experience using this methodology shows that order of magnitude improvements are possible in each of these areas.

The minimally invasive transition methodology eliminates one of the long-standing impediments to the widespread use of first generation software product line ap-

proaches – the adoption barrier. There are two primary parts to this method. The first is to reuse as much as possible and change as little as possible from the current approach in the organization making the transition. This means that rework is minimized and that things remain familiar to developers and managers. The second is to make incremental transitions in such a way that each incremental step offers larger incremental benefits. Incremental return on incremental investment means that a chain reaction starts with just a little investment of time and effort. The ongoing returns are continually larger than the ongoing investments, which eliminates the huge upfront adoption barrier of the early generation methods.

The bounded combinatorics methodology pushes the complexity barrier many orders of magnitude beyond what was possible in the first generation methods. Utilizing modularity and two additional abstraction layers on top of the feature model abstraction, the astronomical geometric and exponential combinatorics of early generation methods can be bounded to small and simple linear combinatorics.

In combination, these new methods offer companies tactical benefits that are large enough to open a new realm of strategic and competitive advantages over conventional development and first generation product line practice methods.

6. References

- [1] Paul Clements and Linda Northrop. 2001. *Software Product Lines: Practice and Patterns*, AddisonWesley, Reading, MA.
- [2] 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.
- [3] See *BigLever Software Gears* data sheet at http://www.biglever.com/extras/Gears_data_sheet.pdf.
- [4] Charles Krueger. *Variation Management for Software Product Lines*, Software Product Lines 2nd International Conference, SPLC 2, San Diego, CA, Aug 2002, Springer-Verlag LNCS 2379, p 257.
- [5] 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.
- [6] William A. Hetrick, Charles W. Krueger and Joseph G. Moore, *Incremental Return on Incremental Investment: Engenio's Transition to Software Product Line Practice*, 9th Intl. Software Product Line Conference, Experience Track, http://www.biglever.com/extras/Engenio_BigLever.pdf.
- [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.
- [11] Charles Krueger and Dale Churchett. *Eliciting Abstractions from a Software Product Line*, in Proceedings of the OOPSLA 2002 PLEES International Workshop on Product Line Engineering. Seattle, Washington. November 2002, pages 43-48.
- [12] Charles Krueger. *Software Reuse*. 1992. ACM Computing Surveys. 24, 2 (June), 131-183.