

Using Separation of Concerns to Simplify Software Product Family Engineering

Charles W. Krueger

BigLever Software, Inc., 10500 Laurel Hill Cove, Austin, TX 78730

(512) 426-2227

ckrueger@biglever.com

http://www.biglever.com

1.0 Introduction

Published proposals and solutions for building software product families rely on some of the most complex, resource intensive, capital intensive, and intellectually demanding software engineering practices developed to date[1,2,6]. For most software engineering organizations, the complexity, cost, and perceived risk are a prohibitive barrier for implementing formal software product family practices.

By applying *separation of concerns*, we have developed a technology that reduces the time and effort required to build and maintain a software product family[7,8]. This technology is a *software mass customization environment* that focuses solely on the *concern* of managing the variation that exists in a software product family. Within that *concern* we identified four basic tasks that are necessary and sufficient for building and maintaining a product family: characterize the abstract dimensions of variation in product family, characterized where the individual product family members lie along those abstract dimensions, identify the locales of variation in the software realization, and characterize how the abstract dimensions of variation are instantiated at the locales of variation.¹

The software mass customization environment works in conjunction with an organization's existing tools and techniques for building single software systems. The separation of concerns is applied so that the technology is independent of language, operating system, configuration management system, build system, and so forth. Furthermore, it does not depend on a domain modeling language, architecture language, or design language.

The environment can be used for all three forms of software family development: proactive (planning ahead for predicted variation), reactive (responding to unpredicted variation), and extractive (reverse engineering variation from legacy software).

¹.Note that these four tasks are the basics of any software reuse technology.[4]

2.0 Motivation

Many of the published proposals and solutions for building software product families rely on domain engineering, reverse engineering, rearchitecting, redesign, reimplementing, complex interacting software processes, system generators, reuse libraries, component assembly validation, and so forth[6]. Because these activities often represent a fundamental shift to heavyweight technologies and complex processes, the timeframes for establishing software product family practices are typically measured in many person-years and many millions of dollars. Furthermore, because of the complexity and extended timeframes, the risk of failure is high.

Contrast this to the fundamental notion of software product family development. Software product family development is, in essence, developing software for a single system along with extensions to account for different, typically small, variations for nearly identical systems in the family. This begs the question, then, as to why the solutions for building software product families aren't as simple as

1. build a single software system, and then
2. build the collection of small variations

Why do we need a major shift to complex and heavyweight software engineering technologies, methods, processes, and techniques?

The answer is that, over the past several decades, we have developed formal tools and techniques for building single software systems (item #1 above), but we have no formal tools or techniques in our arsenal for building and managing a collection of small variations for a software product family (item #2 above). To compensate for this void, software engineers historically have relied on informally contrived solutions such as IFDEFs, configuration files, assembly scripts, install scripts, parallel configuration management branches, and so forth. However, these informal solutions are not scalable. More recently, software product family research has focused some of software engineering's most powerful and complex solutions to managing product family variation.

We believe there is a simpler way to fill this void in our arsenal for creating and managing the collection of variations in a software product family. Using one of computer science’s most powerful principles, *separation of concerns*, we have set out to build the simplest technology that manages the collection of variations in a software product family and at the same time can be used in conjunction with the existing tools and techniques for building single software systems. That is, we want software product family engineering to be a simple extension to single system engineering.

We have adopted the adage that *the right point of view saves 20 points of IQ*. By extending the existing single system tool set with an independent formal technology focused on product family variation, we believe software organizations can achieve the order of magnitude benefits of software product families with an order of magnitude less time and effort than is currently discussed. Rather than talk in timeframes of months and years, we think in terms of what can be accomplished the first day, week, or month.

3.0 Mixing Concerns

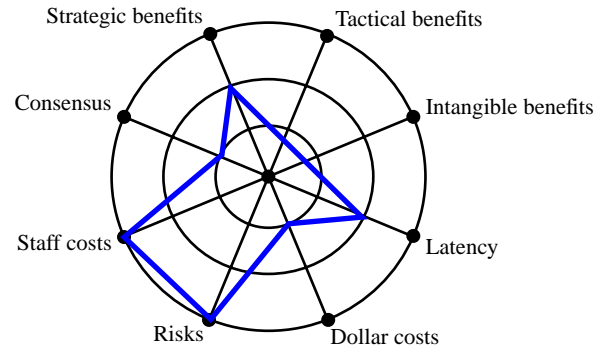
Existing tools and techniques for managing software product families mix other concerns with the management of variation. In this section we very briefly characterize the impact of this mixing of concerns for several of these different tools and techniques. For each, we plot the following eight characteristics on a multidimensional graph:

- strategic benefits
- tactical benefits
- intangible benefits (e.g., improved developer morale, improved customer perceptions, etc.)
- latency (time required to deploy a solution)
- dollar costs
- staff (human resource) costs
- risks (e.g., probability of failure, potential disruption to production schedules, etc.)
- general software engineering practitioner consensus on the perceived value

The three concentric circles are for low, medium, and high rankings (low being the smaller inner circle, high the outer circle). The bold line is the evaluation plot for the tool or technique. Note that the “positive” attributes are along the top and the “negative” attributes are along the bottom, so a “good” option will have a top-heavy graph.

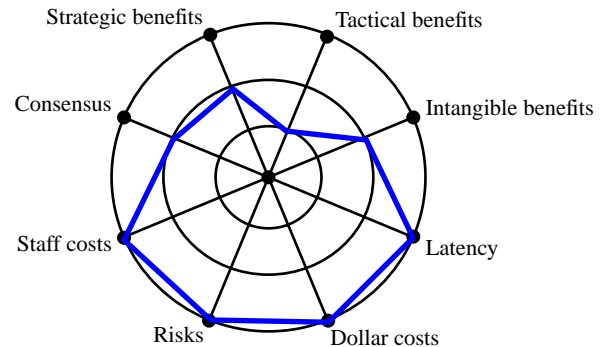
3.1 IFDEFs and Preprocessors

IFDEFs and related preprocessor approaches mix product line logic with runtime application logic. This mixing of concerns does not scale because source code becomes increasingly opaque as multiple preprocessor conditionals are added to a source file.



3.2 Configuration Management

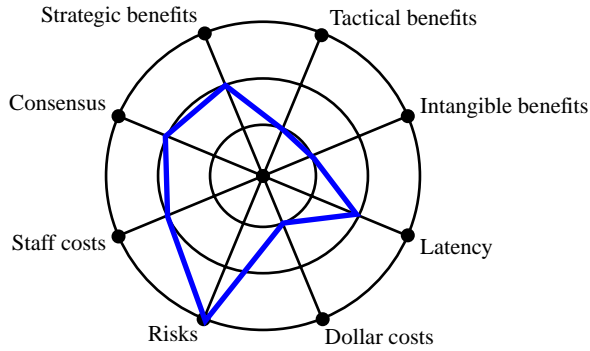
Configuration management (CM) systems are meant to manage system variation over time. Using CM approaches to manage product families mixes concerns of *variation over time* with family *variation at a fixed point in time*. This approach is complex and error prone; it is easy to inadvertently introduce bugs while fixing other bugs.



3.3 Build Scripts

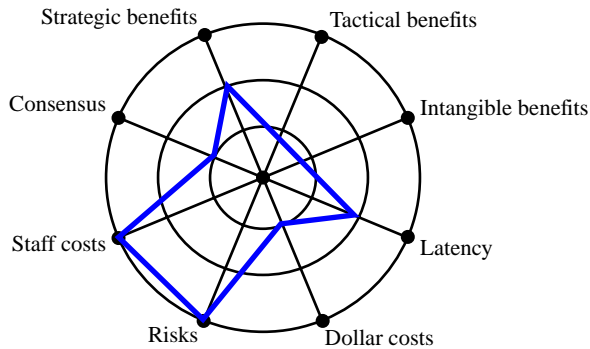
The build script approach is similar to the CM approach. Complexity gets extremely high as the number of products in a product family grows. The dependencies among the different variant and common components are mixed in with the build logic and hidden from the developer, so that impact

analysis for maintenance and evolution is difficult and error prone.



3.4 Config Files

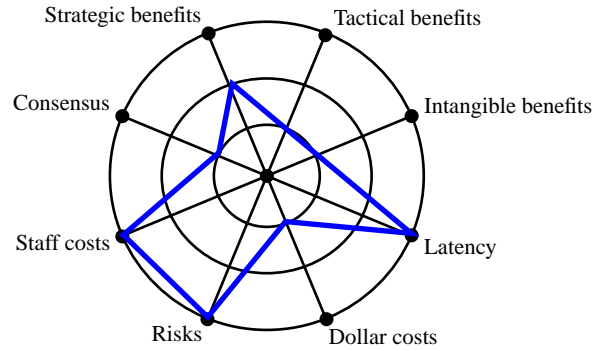
Using external configuration files to influence runtime logic has similar characteristics to the preprocessor approach. The product family variant logic is mixed in with the application logic in the source code. As the product family scales, the product line conditionals in the source code start to obfuscate the application logic, making the code hard to read and maintain.



3.5 Smart Installers

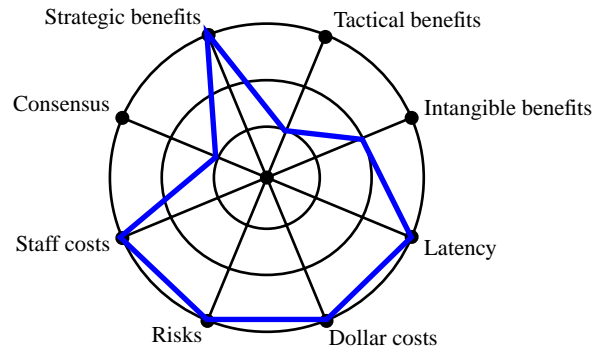
Developing these installers is hard. There has to be a very tight coordination between development, build, CM, and install developers to make this approach work, leading to high overhead. Complexity can get very high. Dependency and impact analysis at the source code level is hard for engineers because the product variation logic is mixed in with the installer logic. That is, there is a flat space of common and

varying software components without any obvious relationship other than what is encoded in a complex installer.



3.6 Domain Specific Architectures

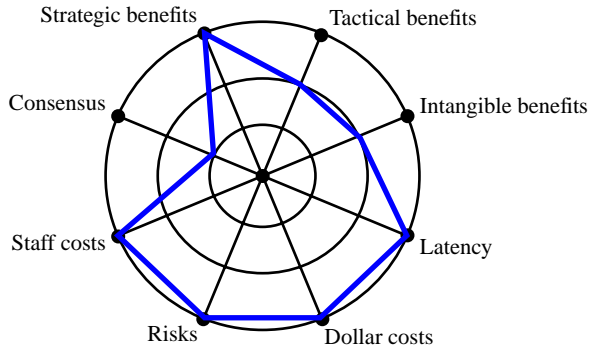
Domain specific architectures provide a formal foundation for software product families[3,5]. However if our stated goal is to manage variation in a product family, domain specific architectures mix the concerns of the common architecture with the variant architecture. With domain specific architectures, reacting to unexpected demands for variants can be difficult due to implications on the common part of the architecture. Furthermore, this approach does not address how the variation at the source level is to be managed (CM, build scripts, config files, etc.)



3.7 Program Generators

Program generators are a formal approach to implementing domain specific architectures as described in the previous section. The profile is therefore similar. In particular, the concerns of the common software in the family is mixed with the concerns of the variant software. Because program

generators are typically complex to build and maintain, the domain and the variation need to be relatively stable.

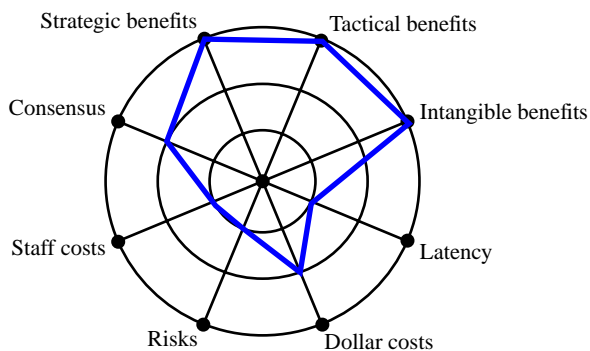


4.0 Separation of Concerns

With a clean separation of concerns, we can manage the variation in a software product family without mixing with the concerns of configuration management, architecture, design, runtime logic, build time logic, installers, and so forth. We leave the existing technology for engineering single systems in place and then add a new point of view for dealing with variation.

4.1 BigLever Software's Mass Customization Environment

We support the four basic tasks that are necessary and sufficient for building and maintaining a product family: characterize the abstract dimensions of variation in product family, characterized where the individual product family members lie along those abstract dimensions, identify the locales of variation in the software realization, and characterize how the abstract dimensions of variation are instantiated at the locales of variation.



This separation of concerns offers the following benefits typically not seen in software product family development:

- Does not require rearchitecting, redesigning, or recoding the existing software. The mass customization infrastructure can be inserted into the existing code base.
- Does not require modifying the content of source files, so the product line logic does not obfuscate the application source code logic.
- Maximizes software reuse by eliminating code duplication and explicitly managing code variants.
- The selection and dependency logic for assembling systems from common and variant pieces is explicitly available to developers rather than hidden in install, configuration, or build scripts.
- Variants of source files or source components are encapsulated together rather than dispersed among different CM branches, source repositories, or build repositories.
- Scales to support hundreds of customized products.
- Works equally well for new development or maintenance
- Supports proactive, reactive, and extractive product line engineering. Proactive means planning ahead for known variation (as with product generators). Reactive means accommodating unanticipated customization requirements. Extractive is taking existing products and bringing them into the product family.
- Separation of concerns makes product line engineering a first class activity rather than a collection of tricks and hacks in conventional technology.
- In addition to managing product family source code, can manage customized test cases, documents, installers, build scripts, graphics, requirements, architecture and design documents, and so forth.
- Independent of CM system and programming language.
- Incremental and low-overhead adoption costs and latency.
- Simplicity means it is easy to learn and use, even by junior engineers.
- Leverage: Small adoption cost, large return on investment.

References

1. Software Engineering Institute. *The Product Line Practice (PLP) Initiative*, Carnegie Mellon University, www.sei.cmu.edu/activities/plp/plp_init.html
2. Weiss, David M., Lai, Chi Tau Robert. 1999. *Software Product-line Engineering*. Addison-Wesley, Reading, MA.
3. Bass, L., Clements, P., and Kazman, R. 1998. *Software Architecture in Practice*. Addison-Wesley, Reading, MA.
4. Krueger, C. Software Reuse. 1992. *ACM Computing Surveys*. 24, 2 (June), 131-183.
5. Prieto-Diaz, R., Arango, G. 1996. *Domain Analysis and Software Systems Modeling*. IEEE Computer Society Press, Los Alamitos, CA.
6. *Software Product Lines. Experience and Research Directions. Proceeding of the First Software Product Lines Conference (SPLCI)*. August 2000. Denver, Colorado. Kluwer Academic Publishers, Boston, MA.
7. Krueger, C. 1997. *Modeling and Simulating a Software Architecture Design Space*. Ph.D. thesis. CMU-CS-97-158, Carnegie Mellon University, Pittsburgh, PA.
8. BigLever Software, Inc. Austin, TX. www.biglever.com