Product Line Engineering Comes to the Industrial Mainstream

Dr. Paul C. Clements
BigLever Software, Inc.
pclements@biglever.com

Abstract. Product line engineering (PLE) is a systems engineering discipline to engineer a portfolio of related products in an efficient manner, taking full and ongoing advantage of the products' similarities while respecting and managing their differences. Managing a portfolio as a single entity with variation, as opposed to a multitude of separate products, brings enormous efficiencies in production and maintenance. This paper shows that PLE has now matured into a repeatable, industrial-strength engineering discipline. We define and explore the concepts central to modern product line engineering, and illustrate how it is becoming applied in two of the most challenging systems engineering domains of all: aerospace and defense, and automotive.

What Is Product Line Engineering?

Product line engineering (PLE) is a way to engineer a portfolio of related products in an efficient manner, taking full advantage of the products' similarities while respecting and managing their differences. By "engineer," we mean all of the activities involved in planning, producing, delivering, deploying, sustaining, and retiring products.

Born in the 1980s in the software field, but now having grown well beyond those early roots, PLE offers large savings observed from engineering the whole family rather than separately engineering each member. Numerous case studies show that exploiting the commonality throughout the products' total life cycles can return substantial improvements in time to market, cost, portfolio scalability, engineer productivity, and product quality [24]; no other engineering paradigm shift has, to our knowledge, brought about results that rival these.

PLE as a Factory

An analogy with factory-based manufacturing serves to illuminate the important concepts.

Manufacturers have long used analogous engineering techniques to create a product line of similar products using a common factory that assembles and configures parts designed to be reused across the varying products in the product line. For example, automotive manufacturers can create thousands of unique variations of one car model using a single pool of parts carefully designed to be configurable and factories specifically designed to configure and assemble those parts.

In PLE, the configurator is the factory's automation component; the "parts" are the assets in the factory's supply chain. A statement of the properties desired in the end product tells the configurator how to configure the assets.

Figure 1 illustrates. This "factory's" supply chain is at the left, in the form of shared assets that are configurable because they include variation points that are expressed in terms of the features available in each of the products. A product specification at the top tells the configurator how to configure the assets coming in from the left. The resulting products, assembled from the configured assets, emerge on the right. This enables the rapid production of any variant of any of the assets for any of the products in the portfolio. Once this production line capability is established, products are instantiated – derived from the shared assets – rather than manually created.

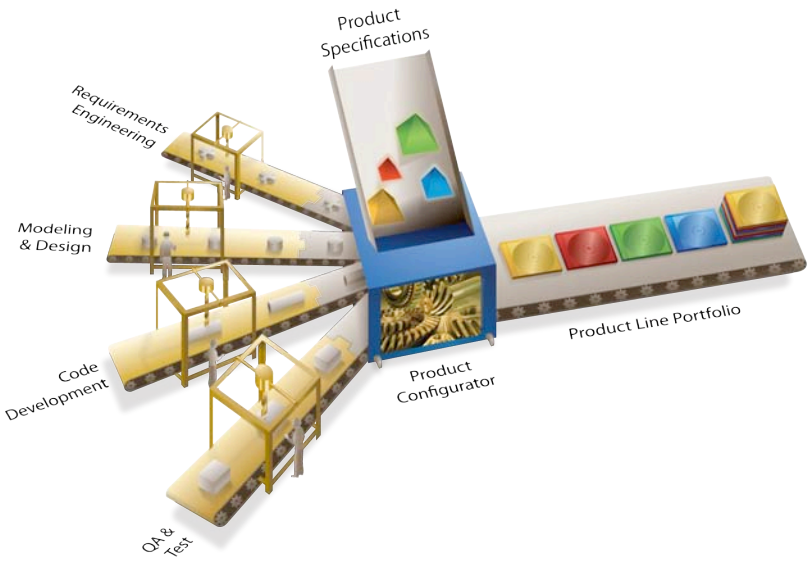

Figure 1: PLE seen as a factory.

The products in the portfolio are described by the properties they have in common with each other and the variations that set them apart. The products can comprise any combination of software, systems in which software runs, or non-software systems that have software-representable artifacts (such as requirements, engineering models, or development plans) associated with them.

In this context "product" means not only the primary entity being built and delivered, but also all of the artifacts that are produced along with it. Some of these support the engineering process (such as requirements, project plans, design modes, and test cases), while others are delivered alongside the thing being built (such as user manuals, shipping labels, and parts lists).

Assets are the "soft" artifacts associated with the engineering lifecycle of the products, the building blocks of the products in the product line. Assets can be whatever artifacts are representable digitally and either constitute part of a product or support the engineering process to create a product. Four kinds of shared assets are shown in Figure 1, but those are just examples. Shared assets can include, but are not limited to, requirements, design specifications, design models, source code, build files, test plans and test cases, user documentation, repair manuals and installation guides, project budgets, schedules, and work plans, product calibration and configuration files, data models, parts lists, and more. Assets in PLE are engineered to be shared across the product line.

PLE Contrasted With Product-Centric Development

PLE stands in contrast to traditional product-centric development, in which each individual product is developed and evolved independently from other products, or (at best) starts out as a cloned copy of a similar product that is then changed to suit the new product's specific needs. Product-centric development takes very little advantage of the commonalities among products in a portfolio after the initial clone operation. In particular, it derives very little benefit from commonality in a product's sustainment or maintenance phase, where data shows most products consume up to 90% of their project resources.

Figure 2 shows a production shop in which *N* products are developed and maintained. In this stylized view, each product comprises requirements, design models, source code, and test cases. Each engineer in this shop works primarily on a single product. When a new product is launched, its project copies the most similar assets it can find, and starts adapting them to meet the new product's needs.

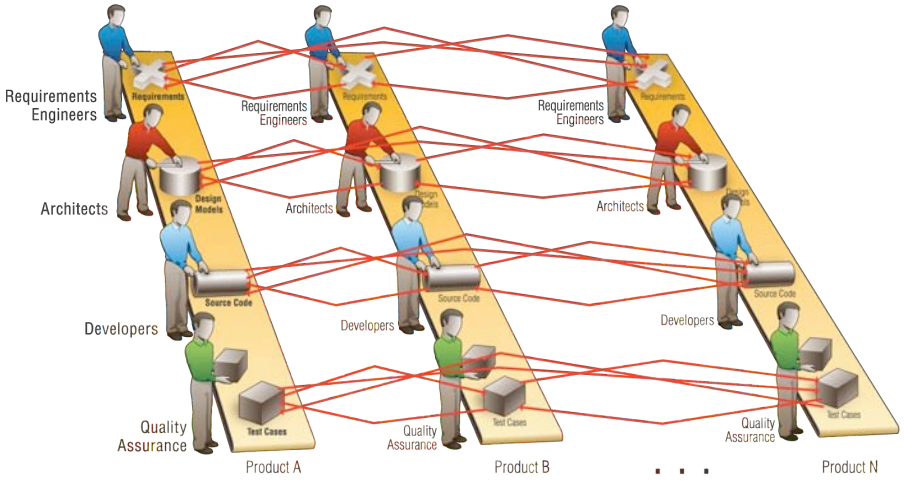

Figure 2: Product-centric development yields $O(N^2)$ complexity

To see how this form of reuse can lead to intractable complexity, assume that a defect is found in Product B, and that the defect is traced to an ambiguous or incorrect requirement in Product B's requirements. The Product B team fixes the error, re-designs as necessary, then fixes the code and test cases before re-deploying Product B. Product B is now healthy again.

But suppose that the defect in Product B's requirements was "inherited" when the Product B team copied the requirements from Product A. Suppose further that the source code for Product N was copied from Product B's (defective) source code, and the test cases for Product *N* were similarly "borrowed" from Product A's (inadequate) test cases.

To really root out the defect from the entire portfolio, each of the *N* product teams should really confer with each of the other *N*-1 product teams. These communication paths are shown in red in Figure 2. This communication obligation imposes an overhead that grows as the square of the number of products. This complexity will quickly overwhelm any engineering staff; in order to get their products out the door on time and on budget, each product team will focus more on

their product silo and less on taking advantage of the commonalities and interdependencies among the other products. The result is divergent product silos, low degrees of sharing, and high duplication of effort across the product silos to fix the same defect multiple times in multiple products, or to independently implement the same enhancements in different ways in different products.

Figure 1 alluded to PLE as a factory, and that analogy can be brought to bear to remedy the $O(N^2)$ problem of portfolio management. In a manufacturing factory, a defective product would not be fixed by one-off repairs to the product itself. Rather, the factory, its supply chain, and the manufacturing process itself would be scoured to find the source of the defect.

So it is with PLE. Rather than fix a defective product, PLE engineers fix the shared asset(s) that need to be modified (perhaps by adding a new variation point) in order to produce the product correctly. Then, the configurator is used to re-generate the product, as well as any other product affected by the changes in the shared assets. Since re-generation has a low and fixed cost, it matters very little whether 2 or 200 or 2000 products need to be re-generated. Thus, fixing a defect, making a systematic enhancement, or carrying out any other kind of portfolio-wide change becomes an $O(N)$ operation.

Early Approaches to PLE

Parnas's seminal paper on program families in 1976 [20] presented the idea that similar programs could be treated as a family rather than as a separate and unrelated set. (It takes only a bit of imagination to see that the concept applies to systems as well as programs.)

Domain analysis, exemplified by the Feature Oriented Domain Analysis (FODA) method [12], provided a way to express the commonality and variations found in a set of systems or products. FODA provided a useful definition of a feature, which is "a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems," and this definition still serves well in the PLE world. FODA led the way to a wide variety of feature modeling languages, which allow domain modelers to express features and their allowable combinations.

The U.S. Department of Defense Advanced Research Projects Agency's STARS project ("Software Technology for Adaptable, Reliable Systems") turned its attention to software product line development in the early to mid 1990s [10]. STARS instilled the dichotomy between domain engineering (the construction of reusable core assets) and application engineering (the selection, application, and augmentation of those assets to build products).

Generative programming [6] involves the use of domain-specific languages in which to specify a product, and a generator to process a product description written in that language to turn out a product. In 1999, Weiss and Lai adopted this approach for a product line methodology called FAST (Family-Oriented Abstraction, Specification, and Translation) [27]. Theirs was the first of several books treating PLE as a topic of study.

Case studies of successful software product lines began to emerge in the mid-1990s. These included STARS demonstration projects, but also commercial successes such as [3]. These studies revealed that successful product lines required more than a technical approach, but also strong management and business acumen as well [19].

Movements began to coalesce to explore product lines from this more holistic approach, first in Europe as a series of Program Families workshops and then in the United States with the creation of the Product Line Practice research program at the Software Engineering Institute and its creation of the Software Product Line Conference (SPLC) series of international conferences (http://splc.net/history.html).

These approaches have yielded a rich legacy of successful product line point solutions [4][18][22][25][26]. However, no approach in these early years rose to the level of what could be called a repeatable, prescriptive, methodological engineering discipline.

Modern or Second Generation PLE

The advent of industrial strength configurators in the early 2000s resolved many of the weaknesses of the early approaches. The presence of reliable commercially available automation that could work at industrial scales rendered much of the early methodological "timidity" moot. The configurators available now strongly support the factory model of Figure 1, and enable the accompanying methodology to do so as well. In the last several years, an alignment of PLE approaches, centered around the factory approach, has emerged that some observers are calling "Second Generation PLE" (2GPLE) [9][15][16]. It has a number of important characteristics that distinguish it from (while building upon) what came before.

Characteristic 1: Features are the lingua franca *to express product differences across the lifecycle*

A feature, to paraphrase [12], is a distinguishing characteristic of a product, usually visible to the customer or user of that product. An example is a function that one product can perform that others cannot.

The concept of "feature" allows a consistent abstraction to be employed when making choices from a whole product configuration all the way down to the deployment of software components within a low-level subsystem in the architecture. In practice, it turns out that stakeholders throughout the entire portfolio's environment are fluent in the language of features. Marketers sell features that customers buy; testers test features; parts are added to support features; software programmers write code to implement features; requirements engineers specify features, and so forth. All of these roles are able to communicate meaningfully in this *lingua franca*, as opposed to the arcane languages of each one's discipline.

Features express the diversity in the product line for a system or subsystem. Feature declarations are analogous to the choices that are available when you buy a new car: Two door or four door? Sport package, luxury package, or economy package? Moon roof? Feature declarations typically express the customer-visible diversity among the products in a product line.

The product line literature is rife with feature modeling languages and constructs, but experience is showing that a very small and simple set of feature modeling constructs suffices for describing all of the necessary feature information for large and very complex product lines [9]. Figure 3 shows a partial example of a feature model in Gears™ (a leading PLE modeling and configuration tool) showing the choices available for an automobile's power door locks system.

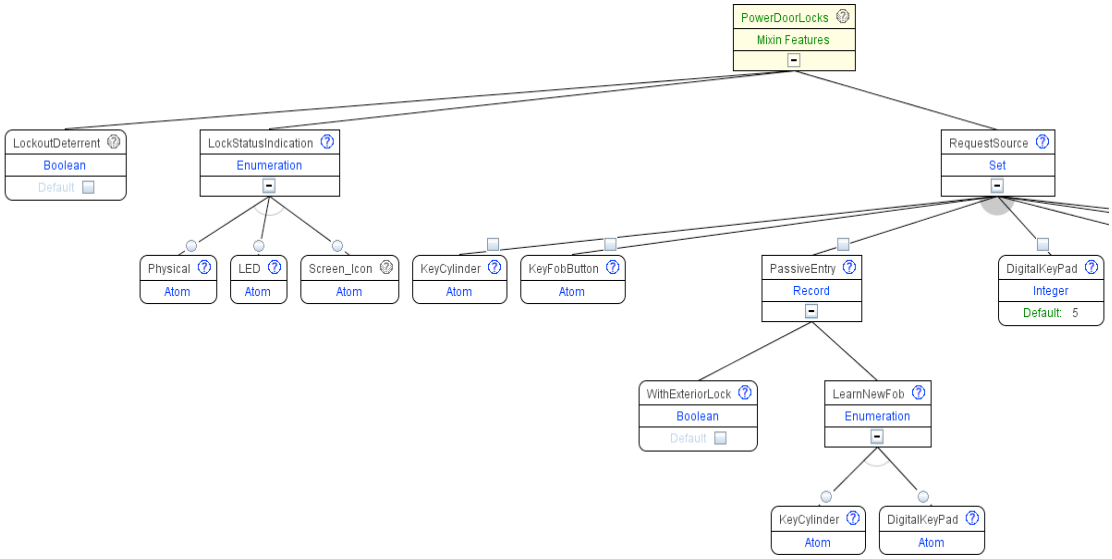

Figure 3: Example of a partial feature model in Gears [13]

Feature assertions. Feature assertions describe constraints and dependencies among the features. For example, feature assertions in Gears express REQUIRES or EXCLUDES relations. They express the constraint that a feature (or combination of features), if present, either requires or excludes the presence of another feature (or combination of features). For example, if we want to make sure that certain features are not available when we're selling our product in a certain region, we could express that constraint with an EXCLUDES assertion between the region feature and the features we want to restrict. Feature assertions are a critical aspect of building feature models; they capture the knowledge necessary to prevent unacceptable (or even illegal) products from being constructed.

Feature profiles. Feature profiles are used to reflect choices from among the available features for the purpose of instantiating a product. A feature profile is associated with a subsystem or a product, and reflects the actual choices you make: Two door with sport package, but no moon roof; or four door with luxury package and moon roof. The values assigned in feature profiles must satisfy the constraints and dependencies expressed by the feature assertions. Feature profiles let us escape the deadly combinatoric complexity of huge product spaces that spring up from even modestly-sized feature trees. Of the astronomical variety available, the set of feature profiles clearly enumerates which (small) set of products are actually of interest.

Characteristic 2: Consistent variation management in artifacts from across the full lifecycle

Second Generation PLE enforces consistent treatment of all shared assets under the production infrastructure, so that a full set of demonstrably consistent supporting artifacts can be systematically generated for each product.

In 2GPLE, assets are designed with built-in *variation points*, which are places in the asset that change depending on the product in which the asset is used. When a product is built, the configurator uses the product's feature-based description to "exercise" these variation points (that is, cause the change in the asset to occur to meet the needs of the product) [14]. Variation point mechanisms comprise:

including or omitted the artifact; choosing one variant of the artifact (from an available set) to use in the product; or making fine-grained choices within an artifact such as including or omitting a section or model element or block of code.

Under this shared-asset-with-variation-points paradigm, the artifacts that engineers create and maintain for the product line are supersets: Each has the content necessary to support any product in the product line. The configurator's job may be seen as exercising the variation points to filter away content until only that needed for the product being built is left.

Variation points are expressed in terms of features, not products. The configurator does its work by comparing feature-based expressions that define a variation point to the feature choices that define a product. Hence, the assets are configured to support feature selections; the supersets become product-agnostic. Among other benefits, this makes adding a new product to the portfolio exceptionally easy.

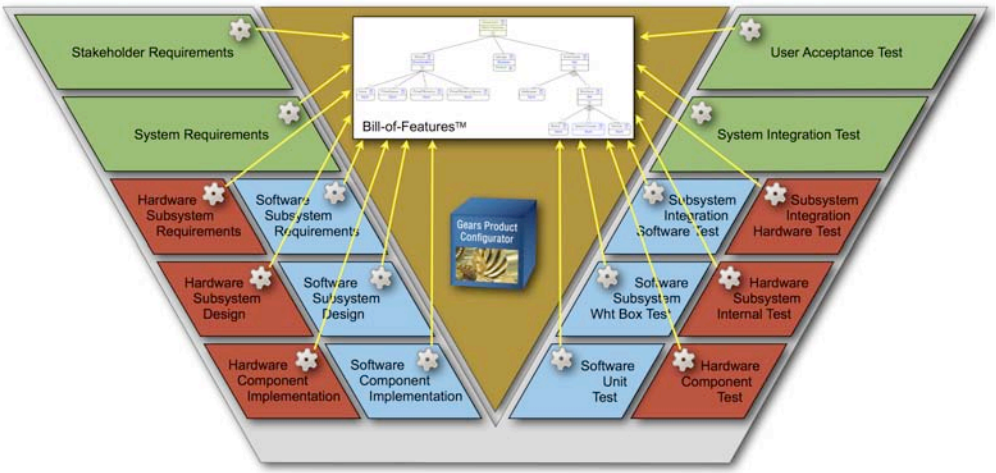

Figure 4: V-model for system engineering, re-cast for product line engineering

Figure 4 shows the classic V-model for systems and software engineering. Each phase is augmented by the addition of variation points (indicated by the gear symbol) to the artifacts native to that phase. A Bill-of-Features for a product corresponds to the feature selections within the feature profiles for that product. The yellow arrows illustrate that all of the variation points in all of the artifacts across the full lifecycle are synchronously and consistently configured according to the single consolidated collection of feature selections in the Bill-of-Features.

Characteristic 3: CM that maintains assets, not products or asset instantiations

Under the 2GPLE discipline, the full superset of available PLE assets (and not the individual products or systems) are managed under configuration management (CM). A new version of a product is not derived from a previous version of the same product, but from the shared superset of PLE assets themselves.

Previous approaches to configuration management (CM) for product lines adopted a "multi-dimensional" approach, claiming that CM for product lines requires CM for core assets and CM for products, and also stating that "CM for product lines is therefore more complex than it is for single systems" [23]. In fact, a key tenet of

2GPLE is to reduce the complexity of product line CM to that for single products, and much less than that for a suite of separately-managed products.

Under the factory paradigm, any defects are fixed in the shared assets, not the products. The affected products can then be re-generated.

Characteristic 4: Supporting product lines across organizational boundaries

Organizations, even small ones, often have separate divisions that work to support their product line. Large organizations almost certainly do. These divisions may be geographically or organizationally isolated from each other. In this case, it is impractical to expect everyone to work on the same feature model, the same set of shared assets, and so forth. Certainly having one global collection of feature declarations for an entire production line is impractical when features may number in the hundreds or even thousands. Subsystem engineers have no interest or need to see all of the feature diversity in other subsystems. For example, engineers for an automotive transmission system do not need to see feature abstractions that capture the diversity in the entertainment or GPS navigation system. It makes no sense to comingle them.

It makes much more sense to modularize the feature model in a way that corresponds to the organizational structure of the enterprise. Although these structures can change over time, they make an excellent starting point and let the organization begin to adopt PLE using familiar units.

For example, an automotive vehicle is composed from combinations of dozens of subsystems, all the way from the engine and transmission down to the subsystems that defog the mirrors and heat the driver's seat. Each of these subsystems has features of its own, which allow a vehicle team to pick and choose in order to define a car. In this way, an automobile (like many complex systems in product lines) is managed like a system of systems, and modeled as a product-line-of-product-lines. This lets engineers work largely independently within the confines of their own organizational units and domain expertise.

Characteristic 5: Industrial-Strength Automation

The last ingredient in 2GPLE is a configurator like the one shown in Figure 1, employed to maintain configurations, and translate feature profiles into assets with their variation points exercised in prescribed ways.

In the early days of PLE, the means of production could be manual, but for product lines of any size or complexity or frequency of change, manual production is impractical; some form of automation is required.

Today there are special-purpose PLE tools called configurators that tie variation points to a central feature model for the entire product line, and provide a set of mechanisms for defining and exercising the variation points in all kinds of assets. Examples of such tools include Gears [13], pure::variants [2], XVCL [11], Dopler [7], and more.

The tooling needs to be able to support the construction and management of feature models (including feature declarations, assertions, and profiles), shared assets of all kinds and their variation points, support hierarchical production lines, and represent the logic that maps from feature choices to asset instances. Further, it needs to either

provide version control for the models and artifacts or (even better) work seamlessly on top of the user's own choice of change management system. It needs to enforce the assertions it has captured, and have a user interface that scales for work across organizations with thousands of engineers whose experience working with engineering tools will likely vary greatly.

PLE in Aerospace and Defense

The aerospace and defense (A&D) sector comprises companies and organizations that make and procure defense products, aircraft, spacecraft, and related systems. One of the most challenging application domains in all of engineering, A&D systems are often at the cutting edge of technology with stringent and challenging requirements. Many of the systems are safety-critical, with exacting, expensive, and (to date) individual-product certification procedures attached.

In the context of Department of Defense procurement programs, a concern is that no Program Office will spend its acquisition dollars to build generalized shared assets that can be used in other programs. Another concern is that the PLE payoff is too far in the future, outlasting the appointed tenure of any acquisition official who, to adopt PLE, must choose to absorb the up-front cost on his watch so that his successor can enjoy the benefits.

However, there are compelling examples appearing that show how PLE is bringing its proven benefits to Program Offices as well as contractors [4].

US Army Live Training Transformation. In 2010 General Dynamics teamed with BigLever Software (the PLE technology provider) to create the winning proposal for the US Army's Live Training Transformation (LT2) family of training systems. Significantly, this contract was the first U.S. Army contract focused specifically on product line engineering as a required part of the solution.

The LT2 training and testing systems portfolio includes live, virtual, and constructive training packaged in embedded and interoperable products that are fielded and used throughout the world. Examples of the many types of training systems in the LT2 family include Military Operations on Urban Terrain (MOUT), Maneuver Combat Training Center (MCTC), instrumented live-fire range training, and various Joint (that is, inter-Service) training systems.

LT2 has long been a true software product line, using first-generation approaches. In 2010 the program made the transition to 2GPLE. LT2 shared assets include the open architectures, common software components, standards, processes, policies, governance, documentation, and more, all leading to a common approach and frameworks for developing live training systems.

The commonality behind LT2 facilitates the rapid development of new products, but also ensures that products across the LT2 product line can communicate and interoperate with each other. This is important because large training exercises need to employ different kinds of training systems working together. The LT2 product line makes use of plug and play components and applications that are common between products, and permits changes, upgrades and fixes developed for one product to be applied to others. This concept provides the inherent logistics support benefits that derive from commonality, standardization and interoperability including the reduction of total life cycle costs [21].

Maximizing asset sharing has proven to reduce fielding time and minimize programmatic costs, while enhancing training benefits afforded to the soldier. Recognized as the Army's live training standard, the LT2 product line architecture, standards, assets, and common operating environment have been used by more than 16 major Army and Department of Defense live training programs with more than 130 systems fielded.

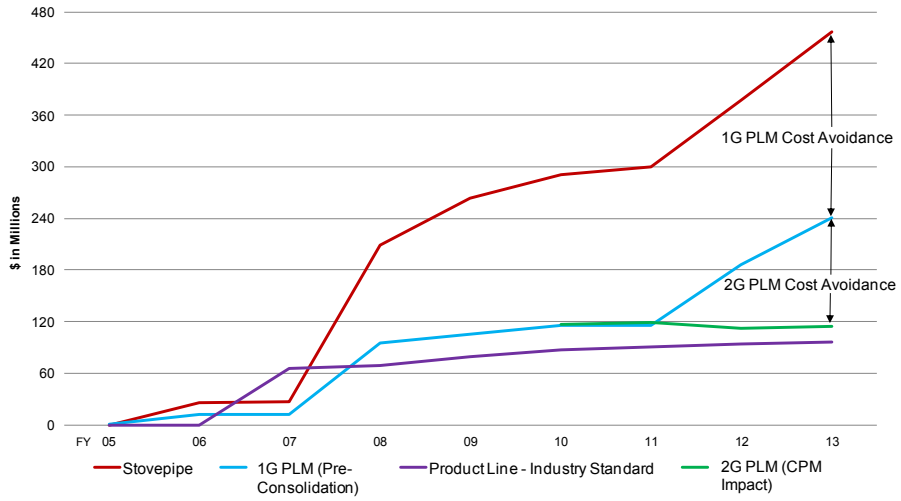

Figure 5: Cost avoidance benefits of product line engineering for LT2

In addition, LT2's 2GPLE approach is exhibiting the following benefits:

- More efficient integration of the Army products by the use of common standards and products to meet training and test requirements
- Compatibility of objective system and products with evolving capabilities
- Wider interoperability before executing subsystem and device production
- Reduced total lifecycle costs to include acquisition, development, testing, fielding, sustainment and maintenance.

This continuing transformation has generated a significant return on investment (Figure 5:) to date within the Army's live training system acquisition portfolio. The first generation approaches generated over $300M in cost avoidance across the development of live training systems to include Combat Training Centers Instrumentation Systems, Home Station Instrumentation Systems, Instrumented Ranges, and Targetry. Their 2GPLE approach, known as Consolidated Product Line Management or CPM in the Army, is projected to save another $200M over the next 2-5 years[1].

US Navy AEGIS Combat System. The AEGIS Combat System is an integrated warfare system deployed on some 100 naval vessels in the U.S. Navy and the navies of key allies across the globe. AEGIS is deployed on deep-water fleet ships, Littoral

[1] These figures are based on industry standard estimates of code cost, and are calculated assuming that post-deployment software support constitutes 70% of development cost and a life expectancy of 10 years. See [8] for a more detailed explanation.

Combat Ships, and (more recently) U.S. Coast Guard National Security Cutters (NSCs) and land-based ballistic missile defense installations. Lockheed Martin's Maritime Systems and Sensors Division maintains the requirements and source code for all product configurations using the 2GPLE paradigm.

In the requirements development phase, requirements are consolidated into a single database (using IBM Rational's DOORS tool) for all stakeholder programs using Gears as the variation engine. This approach avoids redundant efforts and requirements capture when managing program-unique databases. Verification of the requirements is also maintained in the DOORS database.

In the software implementation phase, a master software development repository (CSL) is utilized that contains source files, libraries and configuration files that support multiple product configurations. Products comprise common and unique capabilities such that modifications to common configurations are implemented once and feature-based variation is used to automatically include or exclude each capability from a product.

The combined savings of product line versus clone and own has totaled in excess of $80 million over the past 3 years. Requirements savings for all government agencies has totaled $39 million over the past 3 years. For testing, additional integration testing across multiple programs (instead of one) added 40% in cost to the initial fix, a cost that is gone as well.

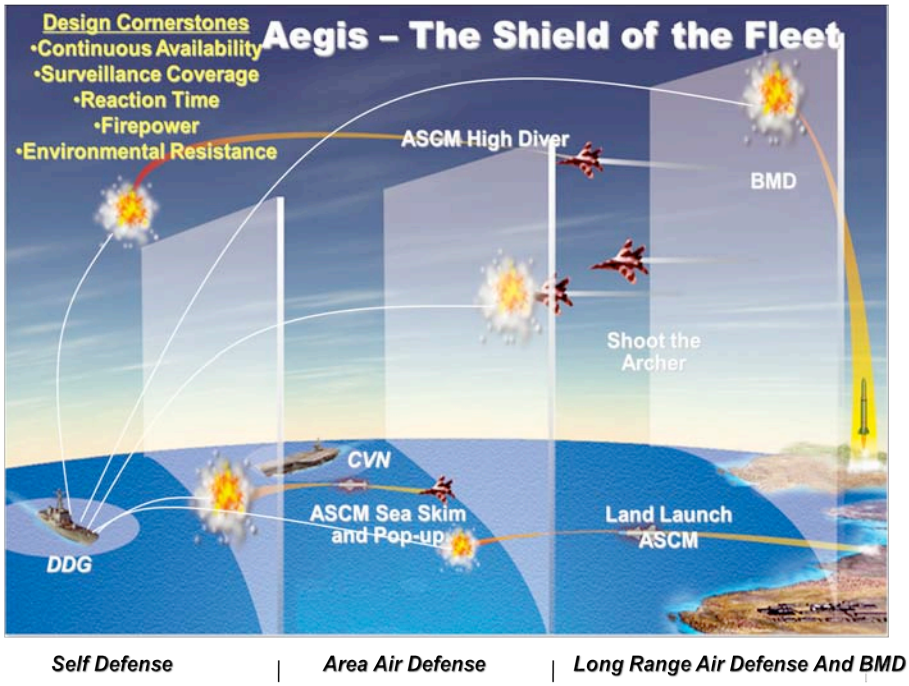

Figure 6: The missions of AEGIS. "ASCM" stands for anti-ship cruise missile. "DDG" and "CVN" signify destroyer and aircraft carrier, respectively.

Savings in new development are being observed as well. Developing an element upgrade for the entire family seems to add, at most, about 10% to the cost of development. This is much less than the cost of cloning and adapting the upgrade (and then testing it) in each of several other programs. Early data collection is trending towards a 40-60% potential reduction in test cases required for new development.

PLE in Automotive

Automotive engineering is a domain rife with complexity. Producing millions of vehicles per year, each one comprising thousands of parts and potentially different from the one just before it on the assembly line, is a feat unmatched in manufacturing. Add to this a slew of next-generation features that require the seamless inter-operation of formerly-standalone subsystems, and that promise more autonomous operation and higher-level user interactions. Today's vehicles must be responsive to the driver's intentions while providing the highest levels of safety and reliability under the most extreme conditions. To meet this challenge, manufacturers have turned to increasingly sophisticated software and electronics, adding yet another layer of complexity as well as the opportunity for expensive and image-bruising defects and recalls. General Motors, among others, is accepting these challenges by adopting the 2GPLE discipline [9][15].

A trend we are observing is that automotive manufacturing is driving PLE to new levels of capability and scale because of complexities that, while new to PLE, are business-as-usual for automakers. Here are some of the most relevant aspects of automotive manufacturing with respect to the 2GPLE discipline:

Lifecycle-wide integration. Full-lifecycle integration is a hallmark of 2GPLE, but automotive engineering is driving PLE tools to substantially increase the size of the engineering ecosystem with which they must seamlessly integrate. A large automotive company will have made tooling choices for each of its engineering artifacts. Perhaps requirements are managed in IBM Rational DOORS, design models in Sparx Enterprise Architect, tests in HP Quality Center, PLM data in Siemens Teamcenter, the owner's manual in Microsoft Word, calibration parameter catalogs in Excel spreadsheets, wiring harnesses in Mentor Graphics Capital, and so forth. To produce the instantiations for individual vehicles, the PLE tooling has to work with each of these tools and preserve the traceability that exists among the artifacts stored in them.

A compelling example of the payoff that integration with a company's tooling can bring is the automatic generation of calibration values. Calibration values are used by automakers that customize their on-board software by using calibration parameters to enable or disable capabilities, or tune the ones that are present. In this setting, feature choices for a vehicle can result in calibration "supersets" being configured to produce calibration sets targeted precisely for that vehicle. The result will be enormous savings in terms of many fewer people doing tedious work, fewer errors to correct in the field as the result of an incorrectly calibrated vehicle, and a vastly shortened process.

Comprehensive constraint capture and enforcement. With thousands of features and feature flavors to choose from, it's critical to have a reliable way to encode and capture all of the knowledge about illegal feature combinations, knowledge that by and large resides in subsystem engineers' heads. Some of it is obvious – no sunroof for a convertible, please – but much of it is esoteric, detailed, and highly specialized. As discussed previously, the PLE tooling has to be able to capture and represent these constraints in an intuitive manner, as well as help document why the constraints are true. Then, it needs to enforce them. Making feature choices for a full vehicle involves many hundreds of selections, and the PLE automation needs to do more than store and enforce assertions. It needs to guide vehicle engineers through the process every step of the way of making feature choices consistent with

the assertions, to prevent any vehicle from being defined and sent to manufacturing that violates any of the constraints.

Support for a product line of product lines (of product lines of…). Features are designed and provided by dozens of different groups. The tooling needs to support the seamless integration of all of their feature models to build a coherent, consistent vehicle. Features in turn need to be supported by technology packages: Choosing a flavor of a park assist feature requires choosing a specific combination of sensors to feed it. Technology choices in turn need to be realized by specific parts, captured in a Bill of Materials. The PLE tooling needs to support knitting together all of the choices that a vehicle comprises, laterally across the organization as well as vertically down through layers of realization specificity.

Options that remain optional right up to manufacturing. In most PLE realms, products are defined with all choices resolved. In automotive, the notion of a *product family tree* comes into play. Vehicles near the top (representing, say, the platform level) have some choices bound but many left open, whereas vehicles near the bottom (for, say, a specific brand, model, sales region, and trim level) have most of their choices selected, but still not all. Options are left open for customers to order, which means that choices need to remain unbound right up until manufacture. Again, the PLE tools have to support this capability.

Option bundles. Options are desirable, but can easily become too much of a good thing. The combinatorics of even a small number of unbound choices can swamp the company's manufacturing capability. Variant and complexity management, sometimes in the form of defining option packages and assigning them sales codes, is essential and the PLE tooling must be able to let product line managers define, analyze, and manage those bundles.

PLE is giving automotive engineering a powerful paradigm shift. Instead of deriving features from a parts list, as has been the mental model in many auto companies, PLE is allowing vehicle engineers to start the design process by considering features first and deriving implementation and realization decisions from those. Features drive parts, not the other way around. This is going to give everyone in the enterprise a common language – the *lingua franca* of features mentioned earlier – as well empower customer-first thinking, and streamline the design and manufacturing process.

Conclusion

This paper has introduced product line engineering as a full-fledged engineering discipline with an established pedigree of providing substantial improvements in development time, cost, and product quality, compared to the old one-product-at-a-time engineering approach. We have also tried to show that modern PLE is not a “boutique” hand-crafted approach, but comes with a repeatable methodology, centered around the factory paradigm and backed up by industrial-scale commercially available tooling. To reinforce this message, we have describe how PLE is being used in two of the most demanding engineering domains: aerospace and defense, and the automotive industry.

Acknowledgments

Thanks to Cathy Martin for a careful and helpful review.

References

[1] Bachmann, F., Clements, P. "Variability in Software Product Lines," Technical report CMU/SEI-2005-TR-01, Software Engineering Institute, 2005.

[2] Beuche, D. Modeling and building software product lines with pure::variants. Proceedings of the 15th International Software Product Line Conference (Limerick, Ireland, September 08-12, 2008). SPLC '08, ACM Press, 358, 2008.

[3] Brownsword, L. & Clements, P. A Case Study in Successful Product Line Development (CMU/SEI-96-TR-016, ADA315802). Pitts- burgh, PA: Software Engineering Institute, Carnegie Mellon University, 1996. http://www.sei.cmu.edu/publications/documents /96.reports/96.tr.016.html.

[4] Clements, P., Gregg, S., Krueger, C., Lanman, J., Rivera, J., Scharadin, R., Shepherd, J., and Winkler, A., "Second Generation Product Line Engineering Takes Hold in the DoD," Crosstalk, The Journal of Defense Software Engineering, USAF Software Technology Support Center, 2013, in publication.

[5] Clements, P.; Northrop, L. Software Product Lines: Practices and Patterns, Addison-Wesley, 2002.

[6] Czarnecki, K., and Eisenbacher, U. Generative Programming: Methods, Tools, and Applications, Addison Wesley, 2000.

[7] Dhungana, D., Rabiser, R., Grunbacher, P., Lehner, K., and Federspiel, C., "DOPLER: an adaptable tool suite for product line engineering," Proceedings of the 11th International Software Product Line Conference (Kyoto, Japan, September 10-14, 2007). SPLC '07, Second Volume, 151-152, 2007.

[8] Dillon, M., Rivera, J., Darbin, R., Clinger, B., "Maximizing U.S. Army Return on Investment Utilizing Software Product-Line Approach," Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC), 2012.

[9] Flores, R., Krueger, C., Clements, P. "Mega-Scale Product Line Engineering at General Motors," Proceedings of the 2012 Software Product Line Conference (SPLC), Salvador Brazil, August 2012.

[10] Foreman, John. "Product Line Based Software Development: Significant Results, Future Challenges," Software Technology Conference (STC) 1996.

[11] Jarzabek, S. and Zhang, H. "XML-based method and tool for handling variant requirements in domain models," Proceedings of the 5th International Symposium on Requirements Engineering (Toronto, Canada, August 27-31, 2001). RE'01, 166-173, 2001.

[12] Kang, K.; Cohen, S.; Hess, J.; Novak, W.; & Peterson, A. "Feature-Oriented Domain Analysis (FODA) Feasibility Study" (CMU/SEI-90-TR-021, ADA235785). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990.

[13] Krueger, C. "The Systems and Software Product Line Lifecycle Framework," BigLever Software Technical Report #200805071r3, 2010. http://www.biglever.com/extras/SplLifecycleFramework.pdf.

[14] Krueger, C. "Variation Management for Software Production Lines." In Proceedings of the 2nd International Software Product Line Conference, San Diego, California, pages 37-48, 2007.

[15] Krueger C., Clements, P., "Second Generation Product Line Engineering: A Case Study at General Motors," in Systems and Software Variability Management: Concepts, Tools, and Experiences, Capilla, Bosch, and Kang, eds., Springer, 2013.

[16] Krueger, C. and Clements, P. "Systems and Software Product Line Engineering," Encyclopedia of Software Engineering, Philip A. LaPlante ed., Taylor and Francis, 2013, in publication.

[17] Lanman, J., Kemper, B., Rivera, J., Krueger, C., "Employing the Second Generation Software Product-line for Live Training Transformation," Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC) 2011.

[18] Linden, Frank J. van der, Schmid, Klaus, Rommes, Eelco. Software Product Lines in Action, Springer, 2007.

[19] Northrop, L., Clements, P. et al. (2009). A Framework for Software Product Line Practice Version 5.0. http://www.sei.cmu.edu/productlines/tools/framework/index.cfm

[20] Parnas, D. L. "On the Design and Development of Program Families," IEEE Transactions of Software Engineering, Vol. SE-2, No. 1, March 1976.

[21] Rivera, J., Samper, W., Clinger, B. (2008). Live Training Transformation Product Line Applied Standards For Reusable Integrated And Interoperable Solutions. Paper No. 483; MILCOM 2008.

[22] Schmid, K. Verlage, M. (2002). "The Economic Impact of Product Line Adoption and Evolution," IEEE Software, Jul/Aug 2002, pp. 50-57.

[23] Software Engineering Institute, "A Framework for Software Product Line Practice, version 5.0: Configuration Management," http://www.sei.cmu.edu/productlines/frame_report/config.man.htm

[24] Software Engineering Institute, "Benefits and Costs of a Product Line," http://www.sei.cmu.edu/productlines/frame_report/benefits.costs.htm

[25] Software Engineering Institute, "Catalog of Software Product Lines," http://www.sei.cmu.edu/productlines/casestudies/catalog/index.cfm

[26] SPLC Product Line Hall of Fame, http://splc.net/fame.html

[27] Weiss, D. M. & Lai, C. T. R. Software Product-Line Engineering: A Family-Based Software Development Process. Reading, MA: Addison-Wesley, 1999.

Biography

Dr. Paul Clements is the Vice President of Customer Success at BigLever Software, Inc., where he works to spread the adoption of product line engineering throughout industry. Prior to this, he was a senior member of the technical staff at Carnegie Mellon University's Software Engineering Institute, where for 17 years he worked leading or co-leading projects in product line engineering and software architecture documentation and analysis. He is co-author of the book *Software Product Lines: Practices and Patterns*, as well as three practitioner-oriented books about software architecture.