The More You Do, the More You Save: The Superlinear Cost Avoidance Effect of Systems Product Line Engineering

Susan P. Gregg
Rick Scharadin
Lockheed Martin
199 Borton Landing Road
Moorestown, New Jersey 08057 USA
+1 609 326 4685
susan.p.gregg@lmco.com
richard.w.scharadin@lmco.com

Paul Clements
BigLever Software
10500 Laurel Hill Cove
Austin, Texas 78730 USA
+1 512 426 2227
pclements@biglever.com

ABSTRACT

Product lines that use automated tools to configure shared assets (e.g., software or requirements or test cases or user documentation) based on product descriptions have long been known to bring about substantial development cost avoidance when compared to clone-and-own or product-specific development techniques. Now, however, it can be shown that the cost avoidance for configuring multiple shared assets is *superlinear* – that is, the overall cost avoidance exceeds the sum of the that brought about by working with each of the shared assets in isolation. That is, a product line that configures (for example) requirements and code will avoid more cost than the sum of code-based plus requirements-based cost avoidance. In addition, we also observe a superlinear effect in terms of the number of products in the portfolio as well. This paper explores why these effects occur, and presents analytical and empirical evidence for their existence from one of the largest and most successful product lines in the literature, the AEGIS Weapon System. The result may lead to new insight into the economics of product line engineering in the systems engineering realm.

Categories and Subject Descriptors

D.2.2 [**Design tools and techniques**]: *product line engineering, software product lines, feature modeling*

General Terms

Management, Design, Economics.

Keywords

Product line engineering, product line economics, systems and software product lines, product line measurement, feature modeling, variation points, product configurator, product derivation, second generation product line engineering, AEGIS.

(c) 2015 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the United States Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

SPLC 2015, July 20 - 24, 2015, Nashville, TN, US.

A Copyright is held by the owner/author(s).

Publication rights licensed to ACM. ACM 978-1-4503-3613-0/15/07…$15.00

1. Introduction

Conventional product line engineering (PLE) economic models count savings as a linear function of the number of products in the product line (e.g. [11]) or as the savings from reuse over the entire product line minus the costs of reuse over the entire product line (e.g., [3]).

However, recent evidence suggests that reality is more nuanced than that. We are measuring cost avoidance associated with each new product added to the product line over and above that product's contributed cost avoidance as an arbitrary member of the product line. In other words, the cost avoidance from a newly-added product are *more* than that from the product last added (all other things being equal).

In addition, there seems to be measurable economies associated with adding new kinds of shared engineering assets (e.g., requirements, code, tests, etc.) to the product line over and above the economies each asset would bring by itself.

Thus, we are observing additional cost avoidance brought about as a result of growing the product line in either (or both) of the product dimensions and the lifecycle phase dimensions. We call this effect *superlinear* cost avoidance, because it exceeds the cost avoidance predicted by the linear cost models that, until now, have been posited in product line economics work.

This paper describes the superlinear cost avoidance effect that is being observed on the AEGIS Weapon System product line, gives analytical evidence that suggests why these superlinear economies are occurring, and provides empirical evidence for their existence.

2. Linear-cost product line economic models

Weiss and Lai posited a simple but useful model of product line economics in 1999 [11] (Figure 1). The model shows that one-at-a-time product development costs grow cumulatively at a faster rate than product-line-based development costs, after an up-front investment to build the product line's reusable assets. It seems fair to say that the model's intent was to convey an intuitive rather than an empirically accurate picture of product line economics, but it is instructive to note that (assuming the products in the product line are roughly comparable to each other in size and complexity) both cost curves are linear. This means that the cost of developing the n^{th} product, under either paradigm, is roughly the same as developing the first, or the 20th, or the 100th.

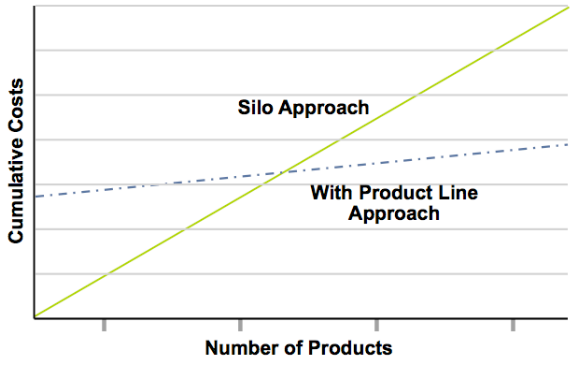


Figure 1 Early product line economic model of Weiss and Lai

Another useful comparison point is the Structured Intuitive Model for Product Line Economics (SIMPLE) [3]. SIMPLE is an economic model for product lines intended to aid in building a business case by providing helpful formulas for calculating true costs and benefits. The cost of a product line development, according to the SIMPLE model, is the sum of:

- C_{org}, the cost of changing the organization to carry out the product line approach instead of silo-based one-at-a-time development
- C_{CAB}, the cost of building the shared assets or "core asset base"
- C_{reuse}, the cost of using the shared assets on each product
- C_{unique}, the cost of carrying out product-specific activities

Figure 2 illustrates a basic SIMPLE formula for the development cost of a product line [4].

$$\boxed{C_{org}() + C_{cab}() + \sum_{i=1}^{n}(C_{unique}(product_i) + C_{reuse}(product_i))}$$

Figure 2 SIMPLE formula for the cost of developing a product line

SIMPLE does away with the assumption that each product costs approximately the same – modelers are free to fill in values for C_{unique} and C_{reuse} for each product independently – but SIMPLE provides no predictive support and no hint that these costs are a function of i itself.

Our purpose is not to criticize these models, but rather to suggest that the effect we report here is previously unobserved.

3. What is the superlinear cost avoidance effect?

We show that the development cost for a product in a product line is, all other things being equal, a function of when the product is added. As the portfolio grows the cumulative cost grows at a slower rate and therefore cost avoidance grows at a faster rate as the portfolio increases in size.

In addition, we posit that economies from adding new kinds of shared assets to the product line also produce this superlinear effect. To see what we mean, suppose the product line begins by including software code (and only software code) in its stable of shared assets. There will be a cost avoidance induced by the product line approach for this "software-only" product line. Suppose that, in a parallel world, the product line had begun with requirements (and only requirements). There would likewise be a cost avoidance induced by the product line approach for this "requirements-only" product line. Now suppose the project includes both requirements and code in its product line; our thesis (and observation) is that the cost avoidance observed *exceed* the sum of the individual amounts, and that this effect continues the more we add shared assets to the product line.

We give the name *superlinear cost avoidance* to these observations, because a graph of the cumulative cost avoidance over the number of products or the number of shared asset types involved is not a line with constant slope, but rather a line with increasing slope.

To elaborate on the different ways in which this effect is being observed, we turn to Figure 3, which lays out three areas of concern or "dimensions" of product line engineering. The multi-product dimension is concerned with the simultaneous development and production of multiple products across the portfolio. The multi-phase dimension is concerned with the various kinds of shared assets that can be configured to support the product line (the figure shows four examples: requirements, design models, code, and test cases). The multi-baseline dimension is concerned with evolving the product line (the products and the shared assets used to build them) over time.

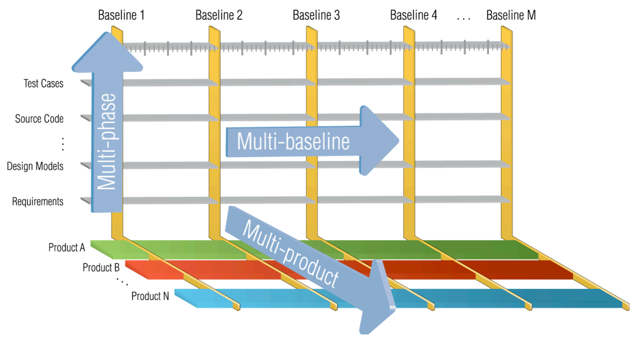


Figure 3 Three dimensions of product line engineering. (Figure © BigLever Software, Inc.)

To summarize, in the AEGIS Weapon System product line we are observing the superlinear cost avoidance effect in the multi-product and multi-phase dimensions of PLE.

4. What is AEGIS?

The AEGIS Combat System is a highly integrated ship combat system. AEGIS cruisers and destroyers constitute the majority of the U.S. surface Navy and will continue to form the core of the surface fleet for the next several decades. The AEGIS Combat System is capable of simultaneous warfare on many fronts: anti-air, anti-surface, anti-submarine, and strike warfare [9]. AEGIS is deployed on some 100 naval vessels in the U.S. Navy, navies of key U.S. allies across the globe, vessels of the U.S. Coast Guard[1], and even land-based ballistic missile defense installations (Figure 4). AEGIS is a system that protects assets from airborne attack from aircraft or missiles. It detects airborne threats, plans how to engage them, and launches missiles to intercept and neutralize them [9].

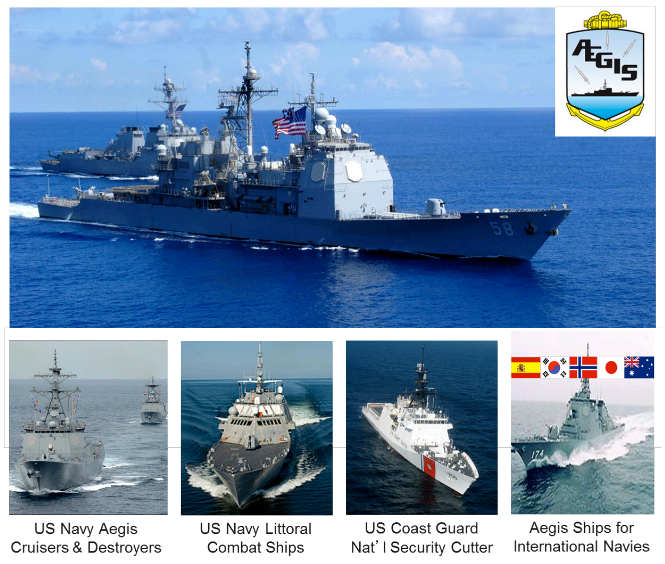

US Navy Aegis Cruisers & Destroyers
US Navy Littoral Combat Ships
US Coast Guard Nat' l Security Cutter
Aegis Ships for International Navies

Figure 4 AEGIS sea platforms include cruisers and destroyers in the U.S. and allied navies, as well as U.S. Littoral Combat Ships and U.S. Coast Guard National Security Cutters. (Figure © BigLever Software, Inc.)

The mission of AEGIS, summarized in Figure 5, includes

- self-defense (protecting the host platform from attack),
- area air defense (for example, protecting a naval task force that includes the host platform), and
- long-range air defense and ballistic missile defense (for example, protecting a geographical area from long-range ballistic missiles).

At the heart of the AEGIS Combat System is the AEGIS Weapon System (AWS), which is a centralized, automated, command-and-control and weapons control system that was designed as a total weapon system, from target detection to kill.

The prime contractor for the AEGIS Weapon System is Lockheed Martin's Mission Systems and Training Division. There, some 1500 people work on the AEGIS program where, among other things, they maintain the over one hundred thousand AWS requirements and over ten million lines of source code used by AEGIS (some 1.8 million SLOC in the last major upgrade alone). Lockheed Martin employs 116,000 people worldwide and is one of the world's largest defense contractors.

[1] Coast Guard vessels employ portions of AEGIS.

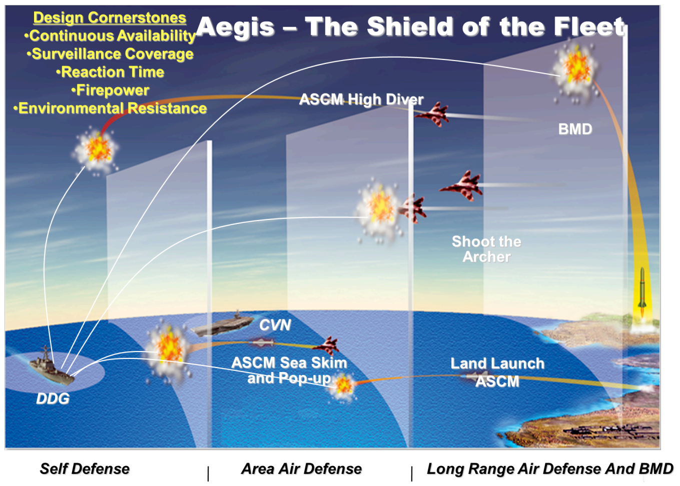
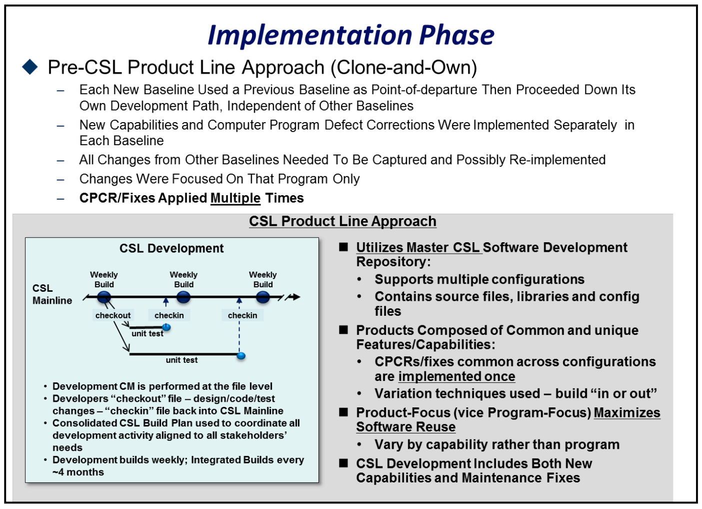


Figure 5 This viewgraph from the AEGIS program highlights the missions of AEGIS. "ASCM" stands for anti-ship cruise missile. "DDG" and "CVN" signify destroyer and aircraft carrier, respectively.

5. The AEGIS Weapon System product line

The products in the AWS product line vary widely depending (among other things) on the platform on which each is hosted. A land-based installation will differ markedly from a naval platform, which themselves differ based on sensors and weapon systems and capabilities. For example, some but not all AEGIS platforms are equipped to shoot down ballistic missiles, a major point of variation indeed.

Like many product lines before and since, the AWS product line evolved from an environment in which members were commissioned, developed, and maintained independently. Copying resulted in a plethora of almost-alike bodies of requirements and code, and a defect shared across ship platforms has to be fixed multiple times at great expense.



Implementation Phase

- Pre-CSL Product Line Approach (Clone-and-Own)
  - Each New Baseline Used a Previous Baseline as Point-of-departure Then Proceeded Down Its Own Development Path, Independent of Other Baselines
  - New Capabilities and Computer Program Defect Corrections Were Implemented Separately in Each Baseline
  - All Changes from Other Baselines Needed To Be Captured and Possibly Re-implemented
  - Changes Were Focused On That Program Only
  - **CPCR/Fixes Applied Multiple Times**

CSL Product Line Approach

CSL Development

- **Development CM is performed at the file level**
- **Developers "checkout" file – design/code/test changes – "checkin" file back into CSL Mainline**
- **Consolidated CSL Build Plan used to coordinate all development activity aligned to all stakeholders' needs**
- **Development builds weekly; Integrated Builds every ~4 months**

- **Utilizes Master CSL Software Development Repository:**
  - **Supports multiple configurations**
  - **Contains source files, libraries and config files**
- **Products Composed of Common and unique Features/Capabilities:**
  - **CPCRs/fixes common across configurations are implemented once**
  - **Variation techniques used – build "in or out"**
- **Product-Focus (vice Program-Focus) Maximizes Software Reuse**
  - **Vary by capability rather than program**
- **CSL Development Includes Both New Capabilities and Maintenance Fixes**

Figure 6 AEGIS implementation management, before and after the product line approach

For example, before the product line transformation, each new code baseline used a previous baseline as a point of departure and then proceeded down its own development path, independently. New capabilities were implemented separately in each baseline. All changes from other baselines needed to be captured and possibly re-implemented. Fixes were applied multiple times. Figure 6 illustrates the before-and-after picture for code.

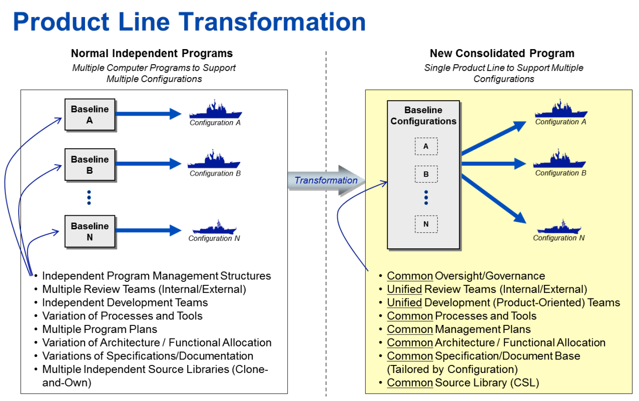

Product Line Transformation
Normal Independent Programs
Multiple Computer Programs to Support Multiple Configurations
Baseline A
Baseline B
Baseline N
Configuration A
Configuration B
Configuration N
Transformation
New Consolidated Program
Single Product Line to Support Multiple Configurations
Baseline Configurations
A
B
N
• Independent Program Management Structures
• Multiple Review Teams (Internal/External)
• Independent Development Teams
• Variation of Processes and Tools
• Multiple Program Plans
• Variation of Architecture / Functional Allocation
• Variations of Specifications/Documentation
• Multiple Independent Source Libraries (Clone-and-Own)
• Common Oversight/Governance
• Unified Review Teams (Internal/External)
• Unified Development (Product-Oriented) Teams
• Common Processes and Tools
• Common Management Plans
• Common Architecture / Functional Allocation
• Common Specification/Document Base (Tailored by Configuration)
• Common Source Library (CSL)

Figure 7 Transformation from independent programs to a true product line approach

Figure 8 illustrates the basic factory approach behind the AWS product line [8]. Shared assets on the left (only a few examples of which are shown) are imbued with variation points. A variation point is a place where a shared asset needs to differ based on whether a feature has been selected or not for a configuration; variation points are defined in terms of features. A variation point might denote a requirement or design model element or code segment or test case that applies, or not, based on feature choices that define a product. A feature profile, describing a configuration in terms of the features it exhibits, is fed to the configurator, which configures the shared assets by exercising their variation points to produce a suite of asset instances specific to the needs of that configuration.

Each product line member has a profile that identifies which capabilities (modeled as features) are included. This method facilitates profiles being updated as capabilities are matured and ready to be deployed in any given configuration.

Gears [1] is a product line engineering tool and framework that powers the factory, enabling users to develop and evolve the product line portfolio. Gears is a feature modeling tool; features describe the capabilities of products; a feature profile is a feature-based description of an individual product. Given a feature profile for a product, Gears also configures shared assets into product-specific instances for that product. Gears integrates with the tools mentioned above, and so engineers can continue to work in tool environments familiar to them. This approach allows users to focus on developing and maintaining a single product line rather than separate, multiple products.

Figure 9 and Figure 10 illustrate the factory for requirements and source code, respectively. The shared assets are on the left of each diagram; configuration based on feature choices results in instances on the right.

Features and profiles are placed under configuration control and managed via an Engineering Review Board (ERB). Changes to existing features/profiles as well as the introduction of new

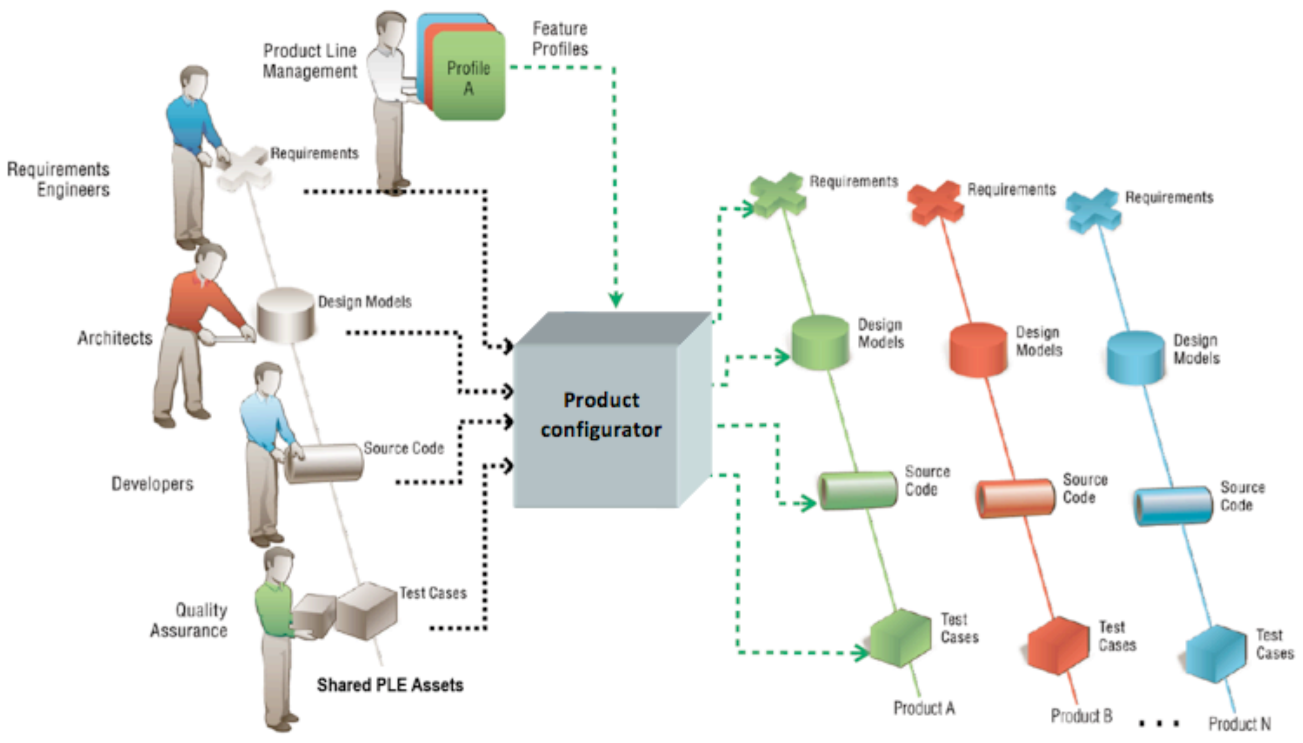


Figure 8 Basic concepts of the feature-based product line factory approach: A configurator configures shared assets (such as requirements, code, and tests, shown on the left) to configuration-specific instances according to the feature profile of the product line member being built. "N" is the number of products in the product line. (Figure © BigLever Software, Inc.)

features and profiles are controlled by the Gears ERB where cross-program considerations are given. A carefully crafted audit process [10] that takes advantage of the Gears tool's audit capability is used to ensure an actuated specification (a) contains the capability in the product configuration based on the feature profile and (b) excludes the constraints or capability that is not part of the product configuration. This same audit is used on the actuated source code.

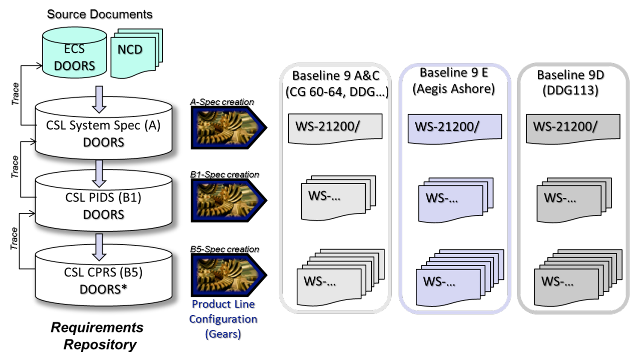


Figure 9 Requirements specs (from high-level to low) are maintained in a single repository, then configured to produce requirements for specific members of the product line.

A key facet of the approach is that a single feature model and a single set of feature profiles apply across all of the shared assets: models, requirements, code, and test cases (with more envisioned for the future).

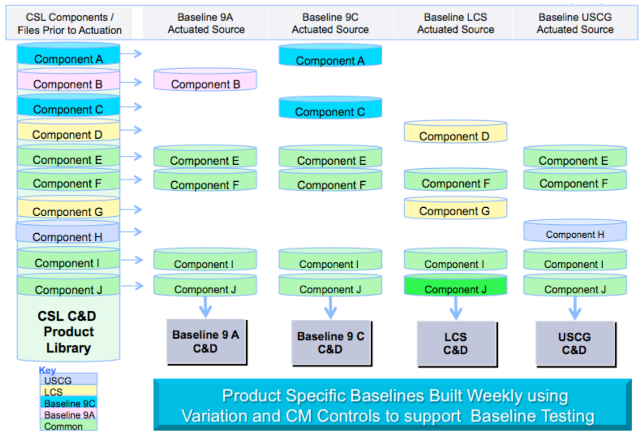

CSL Components / Files Prior to Actuation
Baseline 9A Actuated Source
Baseline 9C Actuated Source
Baseline LCS Actuated Source
Baseline USCG Actuated Source
Component A
Component B
Component C
Component D
Component E
Component F
Component G
Component H
Component I
Component J
CSL C&D Product Library
Baseline 9 A C&D
Baseline 9 C C&D
LCS C&D
USCG C&D
Key
USCG
LCS
Baseline 9C
Baseline 9A
Common
Product Specific Baselines Built Weekly using Variation and CM Controls to support Baseline Testing

Figure 10 The software view of the factory, showing inclusion/exclusion of various components for different configurations. An included component can occur in different forms depending on how its variation points are exercised.

This approach leaves the door open to add new shared assets to the picture in a relatively straightforward manner. Since all shared assets speak the same feature language, as it were, the same set of feature profiles apply across the board. Generally no (or little) additional feature modeling is necessary to configure the new shared asset. As an example, intra-product Interface Design Documents (IDS) are in the AWS DOORS database with Gears variation in place. The IDS use the same features and profiles for configuration as the specs and code, so product configuration versions of the IDS can be produced. The IDS are also traced to the low-level requirements concerning processing, and then in turn to code. In addition to this, the DOORS database is currently being actuated for specific product configurations to produce product-specific training material.

Designing and implementing a product line for all product configurations is a critical tenet. A product architecture must be considered through all phases of a product design, development, and test. In support of product line development, a product architect role was established. This person must have a thorough understanding of the product functional architecture as well as cognizance of programs that will be coming into the AEGIS product line. The architect will also define design considerations to facilitate the entry of new capability into the product portfolio while preserving product core.

In support of the AEGIS product line development, a collaborative cross-program Multi-Baseline System Engineering Integration Team (MB-SEIT) was established to ensure key aspects of system and software architecture. This MB-SEIT has responsibility to ensure proper product line behavior for each of the products.

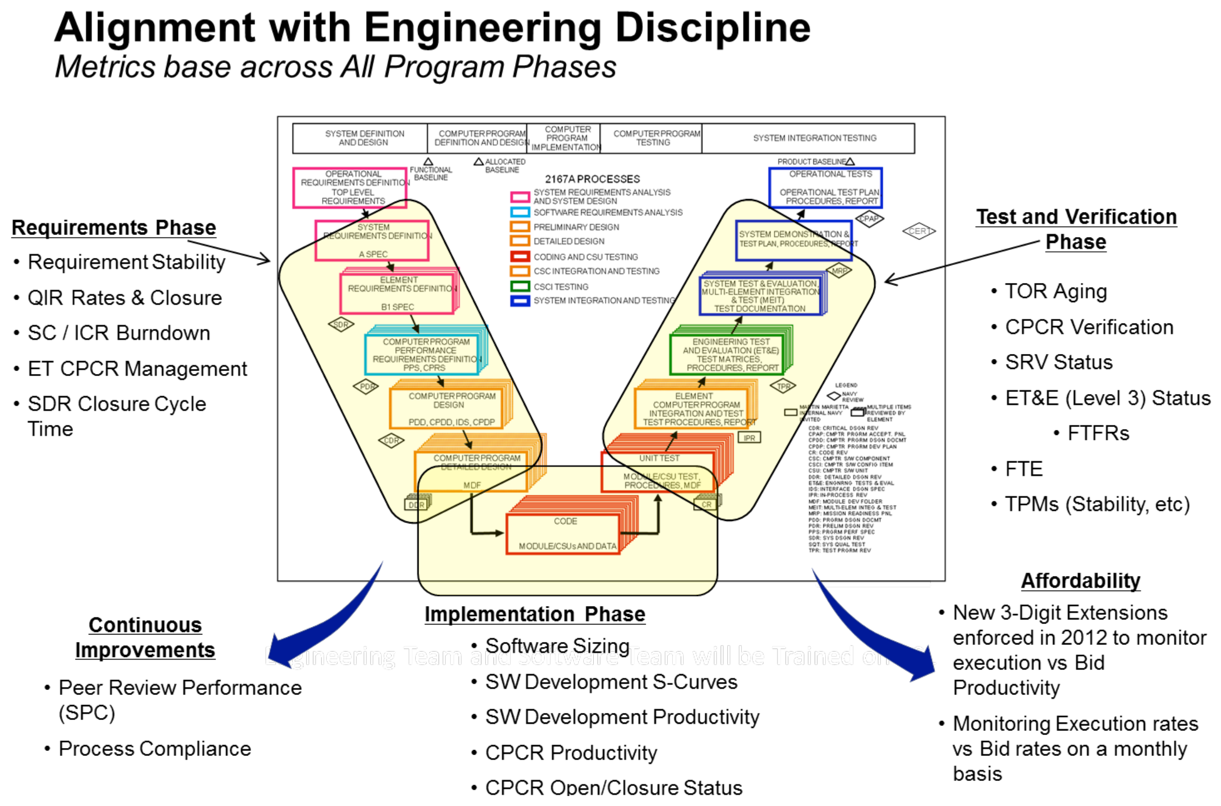
Many more careful policies and procedures have been put in place in addition to the ones mentioned above, to ensure consistency and traceability across shared assets, and to ensure orderly and timely evolution of the product line in a way that is most responsive to customer needs. Customer stakeholders for the product line include different offices of the US Navy, as well as the US Missile Defense Agency and the US Coast Guard. Coordinating the sometimes-conflicting priorities of these stakeholders requires a robust governance structure. Detailed coverage of product line governance is outside the scope of this paper, but described extensively in [7].

6. Economics of the factory

What does it cost to develop a product line under the factory paradigm of Figure 8? Broadly speaking, costs include those listed below. For each, we introduce SIMPLE-like cost function names to refer to them later.

- C_{SAS}: The cost of building and maintaining the shared asset supersets that will be configured to produce individual members of the product line
- C_{FM}: The cost of building and maintaining the feature models that capture the overall distinguishing characteristics among the members of the product line
- C_{FP}: The cost of building and maintaining feature profiles that describe individual members of the product line
- C_{Org}: The cost of re-structuring the organization to take best advantage of the factory approach.

When a new product is added to the family, if its characteristics can be described simply as new combination of already-existing features, then the only cost is adding its feature profile; no changes are needed to the over-arching feature models. Nor are any changes needed to the shared assets, because they were already equipped with variation points to meet the needs of the existing features. In other words, in this case C_{SAS} and C_{FM} (to accommodate the new product) are both zero, while C_{FP} is minimal.



Alignment with Engineering Discipline

Metrics base across All Program Phases

Requirements Phase
- Requirement Stability
- QIR Rates & Closure
- SC / ICR Burndown
- ET CPCR Management
- SDR Closure Cycle Time

Test and Verification Phase
- TOR Aging
- CPCR Verification
- SRV Status
- ET&E (Level 3) Status
  - FTFRs
- FTE
- TPMs (Stability, etc)

Continuous Improvements
- Peer Review Performance (SPC)
- Process Compliance

Implementation Phase
- Software Sizing
- SW Development S-Curves
- SW Development Productivity
- CPCR Productivity
- CPCR Open/Closure Status

Affordability
- New 3-Digit Extensions enforced in 2012 to monitor execution vs Bid Productivity
- Monitoring Execution rates vs Bid rates on a monthly basis

SYSTEM DEFINITION AND DESIGN
COMPUTER PROGRAM DEFINITION AND DESIGN
COMPUTER PROGRAM IMPLEMENTATION
COMPUTER PROGRAM TESTING
SYSTEM INTEGRATION TESTING
FUNCTIONAL BASELINE
ALLOCATED BASELINE
PRODUCT BASELINE
2167A PROCESSES
SYSTEM REQUIREMENTS ANALYSIS AND SYSTEM DESIGN
SOFTWARE REQUIREMENTS ANALYSIS
PRELIMINARY DESIGN
DETAILED DESIGN
CODING AND CSU TESTING
CSC INTEGRATION AND TESTING
CSCI TESTING
SYSTEM INTEGRATION AND TESTING
OPERATIONAL REQUIREMENTS DEFINITION TOP LEVEL REQUIREMENTS
SYSTEM REQUIREMENTS DEFINITION A SPEC
ELEMENT REQUIREMENTS DEFINITION B1 SPEC
SDR
COMPUTER PROGRAM PERFORMANCE REQUIREMENTS DEFINITION PPS, CPRS
PDR
COMPUTER PROGRAM DESIGN PDD, CPDD, IDS, CPDP
CDR
COMPUTER PROGRAM DETAILED DESIGN MDF
DDR
CODE MODULE/CSUs AND DATA
UNIT TEST MODULE/CSU TEST, PROCEDURES, MDF
CR
IPR
ELEMENT COMPUTER PROGRAM INTEGRATION AND TEST TEST PROCEDURES, REPORT
TPR
ENGINEERING TEST AND EVALUATION (ET&E) TEST MATRICES, PROCEDURES, REPORT
SYSTEM TEST & EVALUATION MULTI-ELEMENT INTEGRATION & TEST (MEIT) TEST DOCUMENTATION
MRP
SYSTEM DEMONSTRATION & TEST PLAN, PROCEDURES, REPORT
CPAP
OPERATIONAL TESTS OPERATIONAL TEST PLAN, PROCEDURES, REPORT
CERT
LEGEND
NAVY REVIEW
MARTIN MARIETTA INTERNAL NAVY INVITED
MULTIPLE ITEMS REVIEWED BY ELEMENT
CDR: CRITICAL DSGN REV
CPAP: CMPTR PRGRM ACCEPT. PNL
CPDD: CMPTR PRGRM DSGN DOCMT
CPDP: CMPTR PRGRM DEV PLAN
CR: CODE REV
CSC: CMPTR S/W COMPONENT
CSCI: CMPTR S/W CONFIG ITEM
CSU: CMPTR S/W UNIT
DDR: DETAILED DSGN REV
ET&E: ENGNRNG TESTS & EVAL
IDS: INTERFACE DSGN SPEC
IPR: IN-PROCESS REV
MDF: MODULE DEV FOLDER
MEIT: MULTI-ELEM INTEG & TEST
MRP: MISSION READINESS PNL
PDD: PRGRM DSGN DOCMT
PDR: PRELIM DSGN REV
PPS: PRGRM PERF SPEC
SDR: SYS DSGN REV
SQT: SYS QUAL TEST
TPR: TEST PRGRM REV
Engineering Team and Software Team will be Trained on

Figure 11 AWS metrics

Generalizing this observation, we can see that the cost of adding a new product to the product line is a function of how much new feature content it contains. That unique content must be accommodated in the shared assets (C_{SAS}), and in the features (C_{FM}) and feature profile (C_{FP}) that capture it.

The benefit accrued by adding a new product to the factory can be calculated by comparing the cost of developing and maintaining that product separately compared to the cost of including it in the factory. Under a clone-and-own approach, Lockheed Martin has determined that 35% of the requirements development effort was simply to keep products in sync with each other when, for example, a new requirement was promulgated across all products. A similar amount can be posited for code and tests as well. This cost avoidance is but one example of cost avoidance benefit.

Adding a new shared asset to the factory incurs the expense of building its superset and building in variation points that make the superset configurable to conform to products in the product line (C_{SAS}), and ensuring that the feature model includes the distinguishing characteristics necessary to exercise those variation points properly to produce product-specific instances (C_{FM}). The benefit comes from avoiding the development and maintenance of the shared asset separately for each member of the product line, and keeping those copies in sync as the product line evolves.

As we will see, we are measuring benefits that are growing non-linearly as more products and more shared assets are added.

7. Measuring the superlinear cost avoidance effect

Cost measures for AEGIS have been collected since the product line organizations and all business areas reached steady state product line behavior. Actual per build metrics were collected in both the requirements and software areas to actually measure what work and costs were being avoided.

Metrics are collected and evaluated throughout the product lifecycle (Figure 11). This data is analyzed and opportunities for product improvements are evaluated and implemented. The data is reviewed at monthly product reviews that replace the old program-specific reviews. At the product reviews the product health and all product configuration metrics are discussed. Data collected to data substantiates AEGIS product development as a significant affordability initiative for the government. Real cost is avoided throughout the product life cycle realized as a result of the up-front system engineering effort starting with requirements and following the legacy V-chart.

Figure 12 shows a slide from the AEGIS program contrasting the pre-product-line requirements approach with the product line approach. Notable is the line that says “Typically, 35% of requirements development effort was keeping things in sync” across projects, under the old clone-and-own approach.

That 35% of arguably wasted activity completely disappears under the factory approach of Figure 8, and so is a reasonable starting estimate for what the product line approach might be expected to save. However, the actual system engineering requirements cost avoidance realized to date has been about four times that amount. For software source code the initial estimate of cost avoidance was, again, 35% but actual cost avoidance has been more than double that.

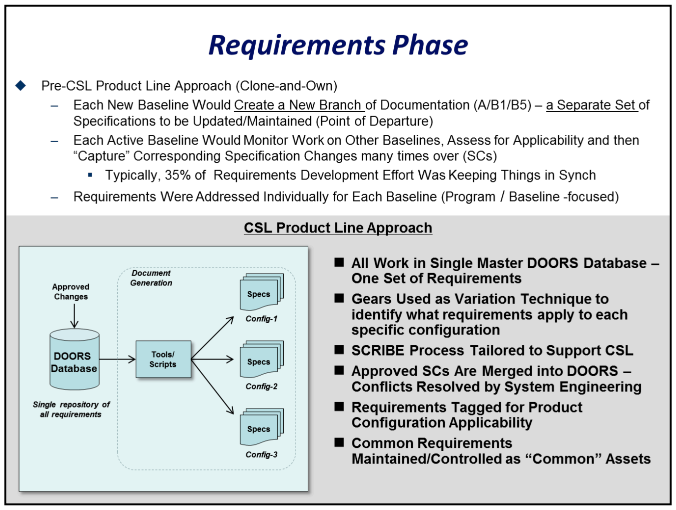
In [7], it was revealed that the product line approach has resulted in $119 million in cumulative cost avoidance over the three-year period from 2011-2013 for the AWS product line, for a per-year average of about $40 million.

Cost avoidance data is measured by capturing the actual requirements and software work performed during each build of the product line, which happens on a scheduled three-times-a-year basis. The cost avoidance calculations use actual measured work converted to dollars. Then the actual work items are re-calculated based on what it would have cost per program using the old clone and own approach.

The latest measures have shown that the cumulative cost avoidance for years 2011-2014 have jumped to $166 million. This means that, whereas we might have expected 2014 to contribute the average $40 million to the cumulative total, in fact it contributed $47 million, or 118% of the norm.

The cost avoidance is measured for each each build through 2014, and is based on requirements and software activities. A build happens at scheduled four-month intervals throughout the year, and reflects the evolution of the product line through the satisfaction of change requests. A build includes new development as well as maintenance fixes, and so could be considered a new member of the product line for purposes of analysis that shows cost avoidance increasing over time. The overall trend seems clear: 2013 showed modest but definite growth in cost avoidance, but 2014 has shown pronounced increases in cost avoidance.

What happened in 2014 to bring about these *additional* economies? The primary answer is that the product line grew, from four to five active major programs.



Requirements Phase

- Pre-CSL Product Line Approach (Clone-and-Own)
  - Each New Baseline Would Create a New Branch of Documentation (A/B1/B5) – a Separate Set of Specifications to be Updated/Maintained (Point of Departure)
  - Each Active Baseline Would Monitor Work on Other Baselines, Assess for Applicability and then “Capture” Corresponding Specification Changes many times over (SCs)
    - Typically, 35% of Requirements Development Effort Was Keeping Things in Synch
  - Requirements Were Addressed Individually for Each Baseline (Program / Baseline -focused)

CSL Product Line Approach

- **All Work in Single Master DOORS Database – One Set of Requirements**
- **Gears Used as Variation Technique to identify what requirements apply to each specific configuration**
- **SCRIBE Process Tailored to Support CSL**
- **Approved SCs Are Merged into DOORS – Conflicts Resolved by System Engineering**
- **Requirements Tagged for Product Configuration Applicability**
- **Common Requirements Maintained/Controlled as “Common” Assets**

Figure 12 AEGIS requirements management, before and after the product line approach. “CSL” stands for “Common Source Library” which, despite its software connotation, is an internal name for the overall product line effort.

The year 2014 thus also saw the cumulative effect of asset re-use across all five programs instead of four. So our cost avoidance formulation, it turns out, is a function of the number of active programs. Exactly what that function is remains to be measured, and discovering it is future work, but at this stage of the exploration it clearly appears to be superlinear.

8. Explaining the superlinear cost avoidance effect

Where do this extra cost avoidance come from that have greatly exceeded even our own hopeful predictions?

We believe the following are the principal causes:

- **Amortization of C_{Org}**: The cost of re-structuring the organization to take best advantage of the factory approach happened over time and was not without false starts and missteps [7]. The AEGIS product line presented the challenge of getting all the people trained on a product line rather than a program specific view. Traditional AEGIS processes were tailored to support product line development. However, that re-structuring has been accomplished, together with all of the processes and procedures and governance activities put in place to effectively run the factory. It is an overstatement to say that C_{Org} will be precisely zero from this point forward, as optimizations will continue to occur, but it is safe to say that it is the case to within a first order approximation. Each new build, then, enjoys the efficiencies purchased with C_{Org} but does not have to bear the cost. As more and more products are added, the fraction of C_{Org} allocated to it continues to decrease. For example, the fourth product would bear one fourth of the cost under a regime that averaged all costs across all products, but the fifth would only bear one fifth.
- **Amortization of C_{FM} and C_{SAS} :** Similarly, the feature models and shared assets are not undergoing wholesale modification, but rather small and incremental evolution. The initial costs are thus similarly amortized over each new member of the portfolio, which bears progressively smaller fraction of the initial cost.
- **Features are the single language to express differences:** Under the factory approach, features are the single authoritative *lingua franca* to express product differentiation and shared asset configuration. This makes adding new kinds of shared assets much easier. Adding a new shared asset entails endowing it with variation points expressed in terms of the *existing* feature model. Just as C_{FM} is amortized across new products, it is equally amortized across shared asset types.
- **Continuing to improve:** The product line team’s organizations have been experiencing continued refinements in behavior and process which is driving higher efficiencies and productivities which will be captured and trended. We find that teams become more gelled and more cohesive over time. Their team structure is common and uniform, and the team continues to fine-tune itself. The overall effect is that of a self-tuning engine with complete standardization and consistency when dealing with each group’s scope of work.

9. Summary and next steps

We have shown that in the AEGIS AWS product line, the cost avoidance per product does not stay the same but grows over time as more and more products are added to the family. Similarly, we have argued that adding a new type of shared asset to the factory is less expensive than adding its predecessors. Cost avoidance is, in other words, a non-linear function of the size of the product portfolio as well as the size of the shared asset portfolio.

This is in contrast to the linear cost models that have been prevalent in PLE up to this point. If future data bears out this trend, it should lend even more weight to the argument for adopting PLE as the engineering paradigm for product portfolios.

The number of AEGIS AWS programs participating during this metrics collection period was five. That is, each build served five different programs. Over the next few years there are expected to be six active programs participating in the AWS product line, which should increase the current ROI percentages and the per build ROI. Future additional ROI metrics collections will include the organizational and test domains.

We are measuring, and will continue to measure team performance in order to quantify our observation that teams are becoming more efficient over time.

During the early stages of transformation to the product line, there were many skeptics who wanted to see the business case for the organizational re-structuring brought about by the adoption of PLE. A return on investment study was performed that analyzed and used the actual pre-product-line metrics and contrasted them against what the predicted product line metrics were expected to be. The system engineering requirements comparison phase predicted at least a 35% return on the number of maintenance requirements that needed to be captured every build. What we are observing now is that this requirement ROI is a function of the number of active programs during each build cycle and is far exceeding our initial expectations.

We hope that other PLE programs will begin to look for this superlinear cost avoidance effect as well, and report it in the literature. We hope that this narrative will add still more weight to the body of evidence showing the enormous cost avoidance brought about by the product line approach, and will encourage others to apply it. AEGIS is, like all aerospace and defense systems, exceedingly challenging in terms of performance, cost pressures, and satisfaction of strict requirements [6]. Yet, we are showing that PLE is not only meeting but is exceeding our best expectations. In this systems engineering PLE case, the more you do the more you save.

10. References

[1] BigLever Software, "BigLever Software's Product Line Engineering Solution," http://www.biglever.com/solution/solution.html

[2] Clements, P., Gregg, S., Krueger, C., Lanman, J., Rivera, J., Scharadin, R., Shepherd, J., Winkler, A. "Second Generation Product Line Engineering Takes Hold in the DoD," *Crosstalk – The Journal of Defense Software Engineering,* vol. 27, no. 1, January/February 2014.

[3] Boeckle, G., Clements, P., McGregor, J., Muthig, D., and Schmid, K., "A Cost Model for Software Product Lines," *Proceedings, Program Family Engineering (5) Conference,* Siena, 2003.

[4] Boeckle, G., Clements, P., McGregor, J., Muthig, D., and Schmid, K. "Calculating Return on Investment for Software Product Lines," *IEEE Software,* special issue on ROI, May/June 2004.

[5] Clements, P., Krueger, C., Shepherd, J., Winkler, A., "A PLE-Based Auditing Method for Protecting Restricted Content in Derived Products," *Proceedings SPLC 2013,* Tokyo, 2013.

[6] Clements, P., Gregg, S., Kreuger, 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,* Jan-Feb 2014.

[7] Gregg, S., Scharadin, R., LeGore, E., and Clements, P. "Lessons from AEGIS: Organizational and Governance Aspects of a Major Product Line in a Multi-Program Environment," *Proc. SPLC 2014,* Florence.

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

[9] Naval Surface Warfare Center, "AEGIS Combat System," http://www.navsea.navy.mil/nswc/dahlgren/ET/AEGIS/default.aspx

[10] Shepherd, J., Winkler, A., Krueger, C., and Clements, P. "A PLE-Based Auditing Method for Protecting Restricted Content in Derived Products," *Proc. SPLC 2013*, Tokyo.

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