ABSTRACT
The Navy is performing product line development within their overall strategy to reduce costs (Emery, 2010). This open architecture product line approach creates commonality across multiple programs and platforms to support faster ship upgrades, cheaper and at a higher quality (US Navy, 2008). In the latest GAO report (2011) on the European Phased Adaptive Approach (EPAA), the GAO noted that the Missile Defense Agency and the Navy had little to no data available to support a life cycle cost analysis of the proposed EPAA approach. In this paper, the author seeks to explore total ownership cost aspects within this product-line approach to determine if current year investment is effective based on the future year costs of maintaining and incrementally upgrading ship platform legacy components (Azani, 2009). The author will examine how the total ownership cost can be impacted at multiple levels of open architecture abstraction, based on the Navy’s existing n-hierarchical n-dimensional family of product lines (Thompson and Heimdahl, 2003) within the sensor domain. The author will illustrate the life-cycle cost differences of retaining the legacy interfaces and software vs. adopting a modular open architecture as part of the overall strategy for the expected life cycle of the next generation ship fleet.

INTRODUCTION
Fundamentally, Navy Open Architecture (OA) is a business strategy to mainstream commercial-off-the-shelf (COTS) techniques, technologies and systems with an integrated technical engineering discipline and approach into the Navy. By reducing the multiple competing architectural infrastructure approaches and embracing a product-line approach, the Navy wishes to “generate true economic efficiencies” in order “to address weapon system affordability, interoperability, and performance for today’s fleet and the Navy after next” (Strei, 2003).

The Navy OA product-line acquisition strategy (Guertin and Clements, 2010) to reduce total ownership costs is consistent with experiences with commercial aircraft fleets. The Boeing Company has been performing product line development for many years as part of its extensive commercial and military aircraft fleet management approach and has well documented approaches when considering legacy system upgrades to align systems along open system product-line approaches.

As part of this approach, PEO IWS is selecting common standard and products in the areas of frameworks, middleware, resource management and operating systems using industry standards that will enable the introduction of common functions across multiple systems and platforms. (Strei, 2003)

As noted in SEI (2011), Azani (2009) and Bosch (2010), product-line development is not only a technical methodology, approach and architecture but also a business practice that transforms organizations. If these best business processes and practices are not followed, significant interdependencies result in a negative effect on the total life-cycle costs. First, the author will seek to understand the best practices and whether these are being followed by the Navy or if legacy business practices and interdependencies still exist.

The Navy has significant challenges for migration of legacy ships and systems toward open systems. There are significant timeline differentials for major ship upgrades and many
existing ships lack the infrastructure to handle many of the new product-lines being produced. COTS systems have to be managed very closely because the technologies change more often than the ships can be improved. This can result in major infrastructure deviations from ship-to-ship over an 8-10 year period which then can result in deploying fragile systems not compatible with the evolving standards.

This author will utilize proven life-cycle cost analysis tools to examine these principles along the n-hierarchical n-dimensional Navy product line to illustrate trade points to explore.

**METHODOLOGY**

This problem becomes an n-hierarchical n-dimensional complexity problem which can be potentially solved utilizing a product-line approach. First by examining the hierarchical approach allows examination of the potential cost relationships from traditional to product-line. Second, by examining how the product-line approach and best practices may create differences in the traditional cost models allows a better understanding of how inter-product relationships drive these costs. Next, an analysis of a singular domain and cost model was performed to examine the cost differences from the product-line approach and how this impacts to overall cost assumptions. Finally, illustration of the effort (cost) differences as the Navy moves along the product-line n-hierarchical n-Dimensional tree, allows some understanding of the relationship of the product-line impacts and potential trades.

**n-Hierarchical n-Dimensional Complexity**

Thompson and Heimdahl (2003) cite evidence that significant advantages to product-lines:

- Cost savings
- Productivity gains
- Competitive edge
- Safer mission critical systems

Even when there may be significant commonalities between members of a family, there still are significant difficulties in applying a product-line approach. Thompson and Heimdahl (2003) and Jones and Northrop (2010) cite many barriers to product-lines:

- Processes
- Organizational acceptance
- Technical constraints
- Lack of a cohesive product family
- Complex interdependencies

Therefore, a better approach is to address these in a multi-dimensional and hierarchical fashion and dealing with near commonalities to extend this approach through a domain. Within Thompson and Heimdahl (2003) and Niu (2009), there is significant discussion on reuse and the use of simple hierarchical product-line family requirement organization and the difficulties in making decisions in this manner. Traditionally, each product-line set of requirements and functionality is broken down until the subset becomes a unique consistent container unto itself, as illustrated in Figure 1. However, this type of analysis may not be sufficient to determine which path may lead to the most effective outcome.

![Figure 1. Traditional hierarchical relationships](image)

**Heuristic:** At the point where the requirements, design, or the expected resultant software/hardware container contains less than 80% commonality (O’Connell, 2011), shown in Equation 1 (the probability that a member is contained with the intersection of sets A and B given the member is also within either set A or set B), is the point where the product-line approach loses significant life-cycle cost savings. Therefore, changes should primarily be considered during activities for transformation,
upgrade, or risk mitigation from sun-setting of critical components (O’Connell, 2011).

\[ |A \cap B| > 0.8 |A \cup B| \]  

(1)

Within a product-line family n-hierarchical n-dimensional relationship illustrated in Figure 2, subsets must be consistent (Thompson and Heimdahl, 2003) which means:

- A subset includes all commonality and variability characteristics of all higher order hierarchical set(s).
- A subset may introduce additional commonality or variability characteristics not present in any of the higher order hierarchical set(s) but these must not interfere or conflict with the characteristic commonality or variability of any higher order hierarchical set(s).
- All commonality and variability characteristics at each level along the n-hierarchical / n-dimensional branches are consistent within each given subset.
- With consideration of n-dimensional aspects, subsets may overlap previously apparent diverging set characteristics and still maintain this consistency.

**Figure 2. n_hierarchical, n_dimensional product-line family relationships**

**Heuristic:** When the subset(s) contain less than three members, \( i < 3 \), the cost effectiveness of a product-line approach becomes minimally effective (O’Connell 2011). Therefore, further break-down should most-likely stop. The rationale is that the extra work associated with building system variances for less than two (existing or planned) products is not cost-effective. There will be some exceptions to this heuristic but these should be treated on a case-by-case basis. Figure 3 illustrates the break-even based on many cases studies.

**Figure 3. Typical product-line break-even**

**Product-Line Interdependency Life-Cycle Cost Drivers**

In Bosch (2010), the final consideration that must be accounted for within the product-line approach showed a single predominant root cause for failure along specifically software product-lines: lack of dependency management. Bosch (2010) and Azani (2009) cite several major activities to assist in managing dependencies.

**Road-mapping of the product-line.** Developing a migration plan with specific goals, tasks, metrics, and milestones is very important (Azani, 2009). Capturing the most-important n-hierarchical n-dimensional characteristics at the top-level for the entire product line and assigning to specific platform releases. Any life-cycle analysis should consider other factors contributing factors such as introduction of new technologies and systems and program interdependency risks (Bosch, 2010).

**Requirements management of the product line.** Defining quality requirement roadmaps for the product-line is critical as shown in Figure 4 (Regnell et.al., 2008) and a driving cost factor to consider within product-line decisions (Nonaka et al., 2007). Analyzing and separating product-specific requirements, rationalizing multiple-product-specific functionality for generalization, and explicitly pushing functionality out of the platform for replacement with commercial or open source components to eliminate bloating or
proprietary functionality, and significantly reduce maintenance (Bosch, 2010 and Thompson and Heimdahl, 2003). This also includes stakeholder involvement that allows assessment of concepts, capabilities, constraints, and strategies (Azani 2009).

**Evolving the software architecture.** The architecting team must systematically change the architecture due to new requirements and response needs (Bosch, 2010). In many cases, the architecture is underspecified and results in costly effort-consuming process coordination (Azani, 2009 and Bosch, 2010) to release the system and any dependent system. Architectural alignments resulted in significant development inefficiencies (Bosch, 2010). The architectural rework can be reflected as shown in Equation 4 (Nonaka et al., 2007):

\[
\Delta Eff_{ar,i}(d_j) = EffDist^{-1}_{wcari}(p) \times WEM_i(t_d) \\
\times DEP_{ki} \times \epsilon
\]

(4)

“\( \Delta Eff_{ar,i}(d_j) \) is the nominal effort of adaptive rework \( d_j \) in months in project \( i \). \( EffDist^{-1}_{wcari}(p) \) is the inverse function of the effort probability function for worst case correction. \( t_d \) is the defect correction time determined by applying a Software Reliability Growth Model (Musa, 1998). \( WEM_i(t_d) \) is the work effort multiplier for the rework for project \( i \) during the maintenance time \( t_d \). \( DEP_{ki} \) is the strength of dependency a continuous function between 0 and 1 with 0 having no dependency and 1 showing a strong dependency. \( \epsilon \) is 1 if \( t_{d,ij} \) is within the period of project \( i \) otherwise it is 0.” (Nonaka et al., 2007). The probability is assumed to be the highest possible value for this paper to illustrate a worst case within the relative relationships. Nonaka et al. (2007) describes the probability distribution as a function of the probability distribution.

It should be noted that Nonaka et al. (2007) discusses adaptive rework due to architectural product-line changes as “corrective maintenance in core assets sometimes brings associated rework to all ongoing product projects that depend on the core assets, to adapt the products to the changed core assets.” Subsequently, each product project \( i \) will be delayed by these change activities if injected during the project. IEEE (1998) defines a term adaptive maintenance which is quite different. Adaptive maintenance is “modification of a software architecture involving change request \( r_j \) in months in project \( i \). \( EffDist^{-1}_{wcari}(p) \) is the inverse function of the effort probability function for worst case requirements change. \( WEM_i(t_{r_j}) \) is the work effort multiplier for the requirements change for project \( i \) during the core asset maintenance time \( t_{r_j} \)” (Nonaka et al., 2007).

Figure 4. Relationship of requirement quality change to product-line cost and benefit

There are key formulas that model this requirements dependency. The first is the probability distribution interval shown in Equation 2 (Nonaka et al., 2007) and the requirements change impact shown in Equation 3 (Nonaka et al., 2007):

\[
Pr(T > t) = e^{(-RC \times t)}
\]

(2)

“\( Pr \) is the probability distribution of the interval between any pair of successive requirements changes \( T \). RC is the mean of the requirement changes per month. RC is the most significant cost driver for nominal cost overruns” (Nonaka et al. 2007).

\[
\Delta Eff_{req,i}(r_j) = EffDist^{-1}_{wcreq,i}(p) \times WEM_i(t_{r_j})
\]

(3)

“\( \Delta Eff_{req,i}(r_j) \) is the nominal effort of the requirements change request \( r_j \) in months in project \( i \). \( EffDist^{-1}_{wcreq,i}(p) \) is the inverse function of the effort probability function for worst case requirements change. \( WEM_i(t_{r_j}) \) is the work effort multiplier for the requirements change for project \( i \) during the core asset maintenance time \( t_{r_j} \)” (Nonaka et al., 2007).
product performed after delivery to keep a computer program usable in a changed or changing environment.” We utilize this model for adaptive rework due to the high dependency of projects on the architectural framework within the product-line development. We also should define that \( NR D_{core} \) is the number of residual defects in core assets which is dependent on product size, complexity, quality, and other factors used for software estimating. This value highly impacts the cost and schedule of all dependent projects.

**Isolating the development.** The product-line development should take place in isolation to minimize costs. However, most teams spend significant time aligning efforts with other development, test, or integration teams (Bosch, 2010). As noted in Nonaka et al. (2007), development can significantly shorten by achieving large-scale reuse but the effort estimations and planning become more complex due to the inter-connected relationships between core assets and products resulting in estimation errors from unplanned work and requirements volatility. Therefore, Equation 4 for adaptive rework would need to be applied here during the development of each product-line project. Additionally, if the software artifacts for reuse are “poor” with a significant number of defects, then the defects will inevitably remain in the core assets as testing alone cannot demonstrate the absence of such. Therefore, the Nonka et al. (2007) model defines the relationship of these defects within the product-line project as shown Equation 5:

\[
\Delta Ef f_{dc-i}(d_i) = Ef f_{Dist_{wcdc-i}}^{-1}(p) \times W E M_i \left(t_{d_i}\right) 
\]

(Nonaka et al. 2007).

**Eliminating the integration costs post delivery.** Integration becomes a significant driving cost in product-line development because while most companies use forms of continuous integration, the configuration management and test infrastructure does not allow full coverage for the required explicit integration and validation here prior to release. A paradigm shift for product-line development has key life-cycle cost reduction principles (Bosch, 2010):

- Teams announce the content of their next release.
- Teams announce interface changes for each release.
- New component releases are backward compatible.
- Teams never build on functionality that’s under development.
- An automated test infrastructure supports the product-line development throughout the life-cycle.

**ANALYSIS**

As noted in SEI (2011), Azani (2009) and Bosch (2010), product-line development is not only a technical methodology, approach and architecture but also a business practice that transforms organizations. If these best business processes and practices are not followed, significant interdependencies result in a negative effect on the total life-cycle costs.

In this analysis, the author first describes the Navy approach as it relates to the domains, interfaces, and complexity with the domain. Next, the author describes the cost-driving assumptions made for each level of the n-dimensional domain examination. Finally, the author will relate each of these whether these are being followed by the Navy or if legacy business practices and interdependencies still exist.
The Navy Fleet and Domains

Per Bosch (2010) within a product-line approach, organizations translate the business strategy into a number of domains of functionality where it seeks significant improvement. The Navy is approaching the objective architecture shown in Figure 5, with this manner for each of the primary domains defined within the PEO IWS Architecture Description Document (ADD) (2009):

- Navigation
- Vehicle Control
- Weapon Management
- Combat Control
- Sensor Management
- Track Management
- Display/ User Interface
- External Communications
- Infrastructure
- Support
- Training
- Computing Equipment

**Figure 5. Navy Objective Architecture Domains**

Within the Combat Systems shown in Figure 6, there are components not explicitly called out as “domains” being addressed within the ADD (PEO IWS, 2009) but as externally provided product-line subsystems that interact with those items produced by PEO IWS. Also notice that any domain may overlap the common core assets, the platform adaptations, and the platform (and non-platform) specific product-lines. Therefore dependencies as defined above will exist between those projects unless product-line methodologies and processes are followed.

- Platform Adaptations
- Sensors
- Communications
- Operational Command and Control
- Vehicles
- Missiles
- Weapons
- Navigational Systems

**Figure 6. Navy product-line relationship to the Objective Architecture domains**

Bosch (2010) states that another tenant of a product-line approach is the lack of a centralized requirement management process. Instead each team associated with a product, evaluates the domain where progress is desired, complements that with its own customer understanding, and tells the organization what it intends to release. The risk that one or more component teams developing related or similar functionality is dwarfed by the inefficiencies in a traditional approach. The Navy currently utilizes a more traditional approach with requirements flow-down into subsystems rather than into the defined domains and specific product-lines within the domain. This results in a more complex requirements process than commercial product-line best practice which focuses on behavior, capability and interfaces. Part of the rationale for the Navy approach, is that timeline and error budgets are tightly coupled to the system performance. By looking at this problem from best practices with a product-line perspective, these requirements can be allocated as characteristic requirements to a domain or
mini-product-line with their own life-cycle at
the top-level without maintaining the entire
requirement baseline centrally or having strict
adherence to these top-level requirements. For
example on a commercial aircraft fleet, several
gine manufacturers can provide the specific
engines for any given platform – they must
adhere to the specific technical performance,
size, weight, power, safety, and control interface
requirements that are defined. The
manufacturer is then responsible for proving
compliance with those characteristics. Their
specific allocation is within their trade space
and eliminates the program dependency of the
body from that of the wing from that of the
engine.

To implement this within the Navy product-line
development, the requirements baselines must
be segmented appropriately to support this
tenant such that only those requirements
absolutely required to be managed centrally
should be managed at the top-level, all other
requirements should be pushed down lower
levels to support the abstraction and remove
these high-order dependencies.

**Domain Interfaces**

Per Bosch (2010) a critical part of the product-
line approach is that product teams (and
components) not only announce the
requirements but all externally visible changes
including interfaces added, depreciated, or
removed. For large product-line projects, a
separate architecture team must focus on
compositionality, backward compatibility and
reducing design erosion by refactoring.

By focusing on these interface changes,
development teams can understand what
changes are required and what functionality
necessary to develop against, eliminating
unproductive coordination hours (Bosch 2010).

This paper examines the interfaces between
domains as well as internal to a domain to
understand and characterize the dependency of
the particular product-line development in
context to show the differences between the
existing system interfaces and the open
architecture product-line approach. For
example, if a sensor system is changed, does this
not only impact the sensor management domain
but also the track management, navigation,
combat control, infrastructure, and display
domains or is the system architected such that
the software or products within those domains
have minimal to no impact?

**Sensor n-Hierarchical Complexity Model**

Various types of sensors both existing and
future are in a single domain: radars (air,
missile, and surface), passive electronic, visible,
electro-optical / infrared (EO/IR), acoustic /
sonar, light-detection and ranging, and multiple
weapon product-line sensors. These existing
sensors have interfaces with a multitude of
legacy systems and evolving future systems.
Each of them would be required to interface or
interact with potentially other sensors for
queueing and to various functions and levels
within each of the other domains.

![Figure 7. Legacy Sensor Interface Complexity](image)

As depicted in Figure 7, the legacy system has
each sensor interacting with each domain or
domain subsystems not exactly in the same way
due to the retention of legacy interfaces. For
example, the SPY-1a may interact with multiple
combat control or navigation subsystems. The
SPY-1d may interact with the same or different
subsystems within that same domain. If those
systems are updated, every single one of the
individual sensor systems that interact with that
particular domain subsystem would then have to
be modified and all the requirements changed
from the top to the bottom of the specification.
tree. This results in the domain interface having an n-complexity to the sensor domain: 23 sensors by potentially 5 interfacing subsystems or 115 interfaces to maintain. This appears to be large but it seems manageable until the analyst looks at the potential combinations that could occur on the ship which reflects a highly complex configuration management program with \(2^{23}\) or \(4e^{34}\) combinations when assuming that any multiple combinations of these sensors with external domain subsystems were possible (Friedman, 2005). Once the Navy factors in multiple domains, this significant value becomes even greater by a significant order of magnitude with \(2^{23}\) or \(4e^{31}\) potential configuration management combinations available. This number seems rather impossible to manage, given singular point-to-point interfaces without trimming and standardization to common Application Program Interfaces (APIs). However, this is the existing legacy upgrade path without moving toward the objective architecture approach in the ADD.

![Figure 8. Domain Sensor Interface Complexity](image)

**If the individual publish/subscribe interfaces are simplified/standardized as depicted in Figure 8, forcing all the legacy sensors to one individual legacy interface to the domain only, and abstracting all other interfaces both internal to the domain and external to other domains, this complexity is significantly reduced to 23 legacy interfaces to the sensors and 9 (or less) API interfaces or 32 to maintain. The configuration combinations would be \(2^{23}\) which is challenging due to having such a large number of combinations but are workable. Tools and processes still are extremely critical to success.**

If the user combined types of sensors together such that as much overlap was possible between sensor types, then significant complexity savings should occur. For example, grouping the SPY-1a, 1b, 1c, 1d and the AMDR together could significantly reduce the complexity if the interfaces and functionality as depicted in Figure 9. This could be examined for commonality to have the minimum functionality absolutely unique to that particular sensor maintained separately (Heese, 2006) (Note: SPY-1b, 1b, and 1d already have a significantly overlapping code and requirements baseline). Given several sensors have the similar/same missions and overarching functionality requirements, this is highly feasible given the upgrade needed for new AMDR sensor introduction.

![Figure 9. Example Domain Functionality](image)

**Upon exploring this route, grouping sensors to maximize the commonality of interfaces and functionality could potentially lead to similar types such as: integrated air and missile defense radar, visible/EO/IR, electronic surveillance, surface, and acoustic/sonar. The complexity of the work then gets reduced by reuse of the same interfaces functionality along the product-line approach. Intuitively, significant cost savings should occur to engineering, development, design, integration and test.**

**Life-Cycle Cost Assumptions**

Because this is a comparative analysis, actual values are not required but parameters are varied to illustrate differential effort. For purposes of estimating the system engineering effort within
the sensor domain, this author has used the following CoSysMo® 2.0 (USC, Jared Fortune, 2009) size parameter estimates. For purposes of estimating ESLOC cost impacts, this author has used CoCoMo II (Boehm, 2000) to evaluate software effort impacts. The tables contained herein define the input parameters to these tools. RC is held to be one per month for this analysis.

**Legacy System Assumptions**

**System-level legacy sensor domain engineering effort is complex.** At the system domain level, the assumptions from Table 1 will be used. Easy, Nominal, and Difficult definitions are related to the tool definitions based on complexity and ease of implementation. Because the sensor domain has an extremely diverse set of sensor systems and a high number of baseline configurations on each ship, several of the parameters would be set to high or very high due to the complexity indicated in Table 2.

**Table 1. Legacy System Size parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Easy</th>
<th>Nominal</th>
<th>Difficult</th>
</tr>
</thead>
<tbody>
<tr>
<td># System Requirements</td>
<td>50</td>
<td>100</td>
<td>15</td>
</tr>
<tr>
<td># System Interfaces</td>
<td>15</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td># Algorithms</td>
<td>400</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td># Operational Scenarios</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code Base SLOC</td>
<td>1,500k</td>
<td>500k</td>
<td></td>
</tr>
</tbody>
</table>

**Legacy Sensor Engineering Effort must be multiplied by the number of sensors.** Each sensor is assumed to have the size parameters shown in Table 3. The cost parameters for the legacy sensor systems are identical to that within the domain. For purposes, of this analysis, it is assumed a nominal productivity utilizing an industry standard average rate of 100 ESLOC / labor month for a mission critical system. Actual values will change based on the amount of auto-generated code from model-based engineering. This is most likely at the system level than algorithmic level that the number will be much higher for the OA because of design for re-use.

**Objective System Assumptions**

**System-level objective architecture (OA) engineering effort is complex.** If the domain were isolated such that the domain contained a simple publish and subscription interface to the external domains, there will be substantial differences within the parameters. The number of interfaces is reduced to a publish interface but the domain must subscribe to most of the other domains. It is assumed that all of the artifacts and requirements are designed for reuse as shown in Table 4 and Table 5.

**Table 2. Legacy Cost Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure</td>
<td>Very Low</td>
</tr>
<tr>
<td>Requirements Understanding</td>
<td>Nominal</td>
</tr>
<tr>
<td>Architecture Understanding</td>
<td></td>
</tr>
<tr>
<td>Application Clarity</td>
<td></td>
</tr>
<tr>
<td>Self-Descriptiveness</td>
<td></td>
</tr>
<tr>
<td>Stakeholder team cohesion</td>
<td></td>
</tr>
<tr>
<td>Personnel/team capability</td>
<td></td>
</tr>
<tr>
<td>Process capability</td>
<td></td>
</tr>
<tr>
<td>Multi-site coordination</td>
<td></td>
</tr>
<tr>
<td>Tool support</td>
<td></td>
</tr>
<tr>
<td>Level of Service Requirements</td>
<td>High</td>
</tr>
<tr>
<td>Migration Complexity</td>
<td></td>
</tr>
<tr>
<td>Technology Risk</td>
<td></td>
</tr>
<tr>
<td>Documentation Required</td>
<td>Very High</td>
</tr>
<tr>
<td>#/diversity of installations/platforms</td>
<td></td>
</tr>
<tr>
<td># of recursive levels in this design</td>
<td></td>
</tr>
<tr>
<td>Percent Design Modification (DM)</td>
<td>100%</td>
</tr>
<tr>
<td>Percent Code Modification (CM)</td>
<td>75%</td>
</tr>
<tr>
<td>Percent I&amp;T</td>
<td>100%</td>
</tr>
<tr>
<td>Assessment and Assimilation</td>
<td>8%</td>
</tr>
</tbody>
</table>

**Table 3. Legacy Sensor Size Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Easy</th>
<th>Nominal</th>
<th>Difficult</th>
</tr>
</thead>
<tbody>
<tr>
<td># System Requirements</td>
<td>100</td>
<td>200</td>
<td>25</td>
</tr>
<tr>
<td># System Interfaces</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td># Algorithms</td>
<td>100</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>Code Base SLOC</td>
<td>250k</td>
<td>100k</td>
<td></td>
</tr>
</tbody>
</table>

The cost parameters in Table 5 for the objective architecture product-line have changes from the legacy because less design and code would be modified. However, the cost parameters are being captured on the high side of a typical
development re-architecting effort. Normally, the DM value would be much lower because the modifications would be captured at a lower level. A simplifying assumption is that system architecture modification would be reduced by approximately 25% to fully capture major modifications within the system.

Table 4. OA System Size Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Easy</th>
<th>Nominal</th>
<th>Difficult</th>
</tr>
</thead>
<tbody>
<tr>
<td># Design for Reuse System Requirements</td>
<td>50</td>
<td>100</td>
<td>15</td>
</tr>
<tr>
<td># Design for Reuse System Interfaces</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td># Design for Reuse Algorithms</td>
<td>200</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td># Modified Algorithms</td>
<td>100</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td># Adopted Algorithms</td>
<td>100</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table 5. Changed OA Cost Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level of Service Requirements</td>
<td>Nominal</td>
</tr>
<tr>
<td>Migration Complexity</td>
<td></td>
</tr>
<tr>
<td># of recursive levels in this design</td>
<td></td>
</tr>
<tr>
<td>Architecture Understanding</td>
<td>High</td>
</tr>
<tr>
<td>Tool Support</td>
<td></td>
</tr>
<tr>
<td>Application Clarity</td>
<td></td>
</tr>
<tr>
<td>Self-Descriptiveness</td>
<td></td>
</tr>
<tr>
<td>Structure</td>
<td>Very High</td>
</tr>
<tr>
<td>Percent Design Modification (DM)</td>
<td>75%</td>
</tr>
<tr>
<td>Percent Code Modification (CM)</td>
<td>50%</td>
</tr>
</tbody>
</table>

Objective Architecture Sensor Domain Engineering Effort is introduced. Abstracting the sensor interfaces to within the domain such that the sensor utilizes a publish/subscribe relationship within the domain allows adoption of the requirements, interfaces, and algorithms significantly simplifying the engineering effort. These new parameters are shown in Table 6 with the same cost parameters as in Table 5. If each sensor remained as an individual sensor without consideration of the overlapping functionality, there would be 23 specific interfaces. However, if considering that 5 of these 9 functions in Figure 9 have approximately an 80% overlap for any given sensor category type (assuming potentially 5 sensor types), allows effort simplification by consolidation of effort up and down the n-hierarchical line. The analysis will examine the difference in effort if this were accomplished.

Table 6. OA Sensor Domain Size Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Easy</th>
<th>Nominal</th>
<th>Difficult</th>
</tr>
</thead>
<tbody>
<tr>
<td># Adopted System Requirements</td>
<td>100</td>
<td>150</td>
<td>20</td>
</tr>
<tr>
<td># Managed System Requirements (those unique to each sensor)</td>
<td>50</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td># Design for Reuse System Interfaces</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td># Deleted System Interfaces</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td># Managed System Interfaces (to sensors)</td>
<td>23 specific / 9 category</td>
<td></td>
<td></td>
</tr>
<tr>
<td># Adopted Algorithms (those unique to the sensor)</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td># Managed Algorithms (to sensors)</td>
<td>92 (23) / 36 (9)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLOC Adopted</td>
<td>200,000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLOC Reuse</td>
<td>50,000</td>
<td>50,000</td>
<td></td>
</tr>
<tr>
<td>SLOC New</td>
<td>50,000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7. OA Sensor Size Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Easy</th>
<th>Nominal</th>
<th>Difficult</th>
</tr>
</thead>
<tbody>
<tr>
<td># Adopted Domain Requirements</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># Managed System Requirements (those unique to each sensor)</td>
<td>50</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td># Design for Reuse System Interfaces</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td># Deleted System Interfaces</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td># Managed System Interfaces</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td># Adopted Algorithms (those unique to the sensor)</td>
<td>4 (if 9)</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>SLOC Adopted</td>
<td>50,000</td>
<td>100,000</td>
<td></td>
</tr>
<tr>
<td>SLOC Reuse</td>
<td>50,000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Objective Architecture Sensor Engineering Effort becomes even less complex. Abstracting the sensor interfaces to within the domain such that the sensor utilizes a publish/subscribe
relationship within the domain allows adoption of the requirements, interfaces, and algorithms significantly simplifying the engineering effort. These new parameters are shown in Table 7 with the same cost parameters as in Table 5. There are multiple options presented based on if the sensors were individually treated or grouped.

RESULTS

In this analysis, the author has captured the results of the CoSysMo© and COCOMO II© estimates. Additionally, the efforts deltas based on the product-line formulas are being included.

The legacy system engineering and software effort shown in Figure 10 use the values from Table 1 and Table 2. The overall driving effort is the Delta Requirements at the system level. Because the system is tightly coupled, the probability of requirements change is very high. Because there is a significant coupling of the requirements throughout the system to the lowest levels, this impact is felt throughout the system. Because there is a one-to-one correlation of requirements at the sensor level, these impacts are directly related to each individual change at the system level – multiplying the effort within the domain by the number of sensors (1S = 1 sensor, 10S = 10 sensors, etc.).

![Figure 10. Legacy System Effort](image)

The Objective architecture system-level engineering effort shown in Figure 11 demonstrates effort reduction directly attributed to product-line approach. Equations 3 and 4 above have a coupling factor of 20% versus the original 100%. There is still an impact at the domain level. The impact within each individual sensor system drives the overall cost. Other studies should look at this parametrically as this analysis has been done with the proposed simplifying assumptions.

![Figure 11. Objective Architecture System Effort](image)

Moving toward an objective architecture and removing and reducing the interdependencies is the only proven successful path for product line approaches. There are clear advantages, not only in the System Engineering but also in the software development to support life-cycle cost reductions.

If the Navy embraced the commonality aspects, the effort reduces more when assuming 4 sensors within each category and 5 different categories with 80% commonality as shown in Figure 12. Note that the defect rate within the core assets for each category will go down although the assets are more tightly dependent. Changes remain the major drivers within the assets and as the probability of change becomes less, the effort to maintain becomes less.

Legacy system costs are driven by stove-piped interface complexity. Migration / modification of these point-to-point interfaces has a significant impact when there are n-Dimensional product-line complexity is incorporated.
The Legacy system engineering effort is significantly driven by the management and maintenance of legacy requirements and interface maintenance which are not designed for re-use within each domain product-line. These results reflect one requirements change per month over a two-year period. The rate of change dramatically changes this curve up/down.

The Legacy development effort is impacted by the lack of a cohesive architecture that supports a product-line approach. The amount of work exponentially increases as more n-Dimensional aspects are incorporated (Figure 10). If the product-line architecture and development team fail to follow these practices outlined, the effort quickly escalates (Figure 13). While half of the legacy effort, it is also magnitudes higher than the optimum from Figures 11 and 12. The downside of the product-line approach is the retesting of changes for the impacted products. Optimally, the more commonality the better in development and maintenance but retest and certification must be addressed within this overall trade space.

By directly comparing the total efforts between the three approaches in shown in Figure 14. Point to point legacy interfaces within certain domains is no longer cost effective based on where the systems are within their life cycle.

The break-even point for typical software product line approaches is the creation of two-three similar code bases. As shown in Figure 15, with our assumptions, the category approach breaks even when there are between 2-3 categories with 4 sensors per category. Commonality within each category was assumed to be 80%. The driver became the delta rates from software dependencies within the category which was higher as previously shown in Figures 10 through 11.

The break-even point along the n-hierarchical product line approach to achieve the greatest savings is a function of the number of similar systems and functions within the domain that interact both within the domain and external to the domain. In this particular analysis, the functions highlighted in Figure 9 were assumed to have a commonality of 80% within a single
category and 20% uncommon within the single category (thereby multiplying the code/requirements by 4 which was the assumed number of sensors within the category). This analysis upheld the two product-line heuristics.

Thi

This analysis upheld the two product-line heuristics.

Therefore to maximize cost effectiveness, the Navy must extract the requirements and functionality down to the level necessary such that each level’s capability and behavior is consistent with each lower hierarchical level.

**CONCLUSION**

The most important cost driver remains the rate of change for requirements and defects.

- Defect rates predictably go down when utilizing core software components.
- The rate of change at lower levels reduces significantly with product-line approach by orders of magnitude (Figure 14).
- Requirements that embrace product-line commonality and reuse reduce impact.

The Legacy development effort exponentially increases as more n-Dimensional aspects are incorporated within a domain (Figure 10).

Failure to fully embrace product-line tenets impact the cost negatively but remains less from the traditional approach (Figure 13 shows about 50% savings) but is an order of magnitude higher than the optimum approach (Figure 14).

Capturing not only the domain-level but unique sensor category-level requirements and design reduces efforts (Figure 12).

- The break-even is 2-3 common components to fully migrate to MOSA and eliminate legacy code infrastructure (Figure 15).
- Unique algorithms and design should be kept at the lowest-level possible to maximize cost effectiveness.

Future studies should consider use of dynamic models within specific Navy domains to address re-test and certification costs associated with the product-line approach.

**REFERENCES**

Azani, C. “A Multi-criteria decision model for migrating legacy system architectures into open system and system-of-systems architectures.” Defense A R Journal, 16.3 (2009). Defense Acquisition University, Expanded Academic ASAP.


Niu, Nan, “Extractive product line requirements engineering” (PhD diss., University of Toronto (Canada), 2009). AAT NR61130.


Thompson, Jeffrey M. and Heimdahl, Mats P.E. “Structuring product family requirements for n-dimensional and hierarchical product lines.” Requirements Engineering, Volume 8, Number 1 (2003): 42-54. doi:10.1007/s00766-003-0166-0

University of Southern California, Dr. Jarrod Fortune. CoSysMo is a copyrighted product of the University of Southern California and Dr. Ricardo Valerdi. Version 2.0(2009) utilized is copyrighted by Dr. Jarrod Fortune.


ACKNOWLEDGMENTS

The author wishes to thank Dr. Jerrod Fortune and Indrajet Dixit at the University of Southern California and colleagues at The Boeing Company for their thoughtful comments and encouragement with regard to this work.

Kelley Caudle is the principal author. Student at the University of Southern California, and an employee of The Boeing Company. She previously was a Director at S2 Corporation and CEO/CTO at Digital Operations. As Chief Engineer for the Ground-Based Missile Defense Ground Segment, she was responsible for the technical design, deployment and fielding of the C4ISR fire control and embedded test/training. She is a member of IEEE, AFCEA, WID, NDIA, ASNE, and Sigma Pi Sigma. She has had numerous conference presentations including for the SBA conference, MDA SBIR conference, and Mentor-Protégé conference. Recent publications include Why Don’t You Just Shoot the Moose: Using Common Sense in Every Day Business Practice, “Integrated Air and Missile Defense in a Communications Limited Environment” (AIAA March 2009), “Extending Battle Management for European Defense” (AIAA March 2010), “Multi-national Interface Challenges for Evolving Missile Defense Systems” (AAAF February 2010), and “Evolving Cross-Domain Coalition Information Sharing and Shared Situational Awareness” (International Missile Defense Dec 2009).