Implementing a Product Line-Based Architecture in Ada

Joel Sherrill, Ph.D.
OAR Corporation
4910-L Corporate Drive
Huntsville, AL 35805
(256) 722-9985
joel.sherrill@OARcorp.com

Jennifer Averett
OAR Corporation
4910-L Corporate Drive
Huntsville, AL 35805
(256) 722-9985
jennifer.averett@OARcorp.com

Glenn Humphrey
OAR Corporation
4910-L Corporate Drive
Huntsville, AL 35805
(256) 722-9985
glenn.humphrey@OARcorp.com

ABSTRACT

This paper describes a software component model that encourages reuse in application families by recognizing and leveraging similarities between products within a product family, as well as among product families themselves. By applying a product-oriented view, developers gain insight into the capabilities of the organization's products and can leverage that insight to incorporate common software components across the entire enterprise. This component model is being applied to an existing family of similar embedded systems whose application software is written in Ada. Features of this language will be examined in the context of how they facilitate construction of reusable product line-based components.

Keywords

product line architecture, reuse, risk management, software lifecycle, component architecture, object-oriented, Ada

1. INTRODUCTION

Software developers are faced with a set of complex challenges. Each new system, upgrade, or derivation from an existing system must be developed faster and at a lower cost than its predecessors with fewer defects. The key question for software developers, their managers, and corporate management is how to thrive – not just survive -- in this increasingly difficult environment. A reuse model that leverages commonality between members of a product family as well as across similar product families is an effective way to meeting this challenge. Reusing existing software is a well-known approach to lowering development costs, time to market, and maintenance costs while increasing system reliability. Encouraging reuse within products makes it easier to present a common look and feel across multiple product families, thus minimizing end user training, testing, and system deployment costs.

This paper is based upon experience gained from applying a product line-based reuse to an existing family of embedded applications written in Ada95. The examples are constructed to accurately reflect the product family structure of that original application, but are different from that application.

2. FAMILIES OF APPLICATIONS

A successful product quickly expands into a family of products to increase or maintain market share (sales) by capitalizing on the market success of the original product. A proven model for cost effectively developing and maintaining a product line centers upon three key concepts: the core architecture, the product family architecture, and the individual product.

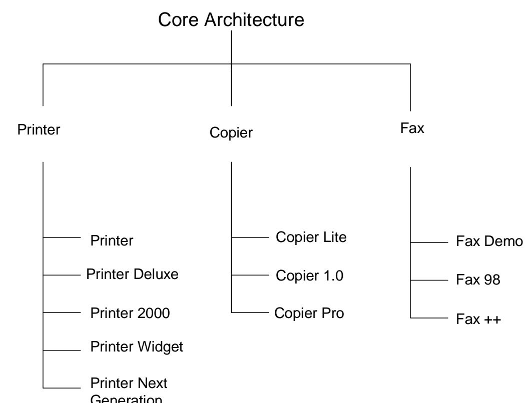

Figure 1 – Product Family Tree

Figure 1 depicts an organization that is responsible for three product families: a laser printer line, a copier line, and a FAX machine line. Each of the product families consists of several variations of the product ranging from low-end, low cost products to high performance versions. Also included are research and development versions demonstrating potential features for future variations.

For each product family, there is a person or group that governs the product family. That group is responsible for ensuring compatibility and managing shared functionality between the individual products. This shared functionality

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page.
To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGAda 2001 09/01 Bloomington, MN, USA

© 2001 ACM 1-58113-392-8/01/0009...$5.00

is maintained and controlled by the governing body that must keep all existing products as well as future products in mind when making changes, or accepting new features. This group tries to recognize the potential for reuse between other product lines within this organization and their own. Maintaining and controlling this reuse between the product lines is accomplished through the core architecture. Impacts to all product families must be considered when accepting changes or features.

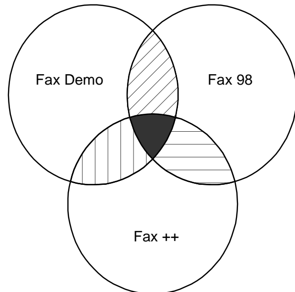

Figure 2 – Functionality Overlap Within a Family

As illustrated in Figure 2, products in the same product family may have similar functionality. This can be leveraged to obtain reuse between those products.

However, it is also possible that products from different product families may be able to share functionality. This shared functionality is the foundation for a core architecture that can be used across multiple products and product families.

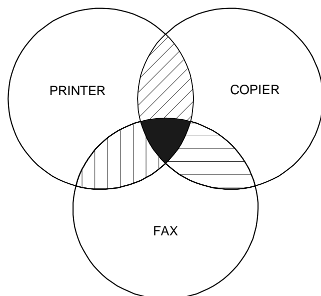

Figure 3 – Functionality Overlap Between Families

In the example product families, a single "print engine" may be shared across all models of all three families. This is represented by the solid area in Figure 3. The hardware community has recognized that reusing the same component in multiple products reduces the cost of that component for all products. This shared print engine is the hardware equivalent of the core architecture concept discussed in this paper.

As product families grow and change, their functionality will fluctuate to obsolescence, technology improvements, or changing requirements. This allows functionality to become reusable where originally it was not. Cross-pollination of features occurs when a feature is developed and then reused by another product. Consider that the FAX machine line wishes to create a new product that has an Internet connection, and is called the iFAX. The product design team begins by examining the functionality that currently exists in the product family's common component set. As a result of this examination, they determine that the product family's common component set does not have Internet communication software. Moreover, the design team believes this functionality has the potential for reuse. The design team for the new product consults with product family and core architecture representatives to see if Internet communication software is under development by any other product or product family.

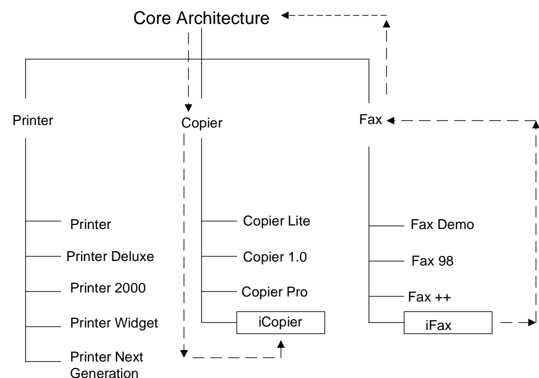

Figure 4 – Family Tree with Internet Migration

Figure 4 demonstrates how after the iFAX product develops this new functionality, it becomes available for reuse. It is subsequently incorporated into the Copier product line when dictated by market demand. This feedback process provides a natural and efficient mechanism for multiple products to benefit from new functionality developed by other product teams.

3. CONFIGURATION CONTROL

As the amount of shared software extends across project families, the reusable source becomes an important asset to the enterprise and should be viewed as a product for the purposes of testing and release management. It is very important to have a defined configuration for reusable and potentially reusable source.

Additionally, the shared source must be managed in a way that allows for individual projects to incorporate new functionality for submission and/or update to a new release. Additionally, it must be recognized that slight modifications are often necessary to effectively reuse a resource. For example, a new project may find code reusable if a table of information is used instead of hard values. This is a relatively simple change, but when and how the originator incorporates this change is not as simple. This section will demonstrate an approach that provides each project the ability to upgrade to a tested, released version of a common

architecture and to submit functionality into the common architecture.

A regression test suite is necessary to verify reusable functionality. By providing repeatable test cases with expected results, the reusable software's behavior can be verified across all platforms. This becomes very important when porting to a new environment or upgrading tools. As the capabilities of the reuse base grow, the importance of robust regression testing cannot be overemphasized. But this does not explain how products can develop functionality and subsequently submit it for inclusion in the reuse architecture along with tests. The feedback process is depicted in Figure 5.

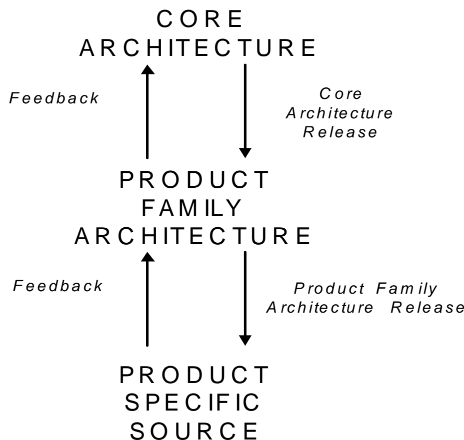

Figure 5 – Feedback Release Process

Consider how a reusable component evolves through the release cycle. At product conception, a copy of the latest release of the product family architecture is installed. The product family architecture comes with a set of tests that may be run to verify the installation and proper functioning of the product family architecture. The product now develops functionality for submission as a reuse component. This functionality should include a set of tests with expected results. The project is responsible for maintaining the functionality until it is accepted into a reusable architecture. If the functionality is submitted to the core architecture and accepted, it will become part of the next release. This release should be incorporated into the product family architecture. Once a new release of the product family architecture is made that includes the new functionality, the product can then upgrade to the newly released version of the product family architecture that now includes the newly developed functionality. This new functionality is now being maintained as part of the core architecture and is available for use by future products both in this product family and within other product families.

Notice the emphasis this discussion placed upon treating the reusable source as a product with regression testing and releases. This is very intentional since it formalizes each product's view of the reuse base. The goal is for individual product teams to view the Core and Product Family Architectures as products and treat them as such for the purposes of testing and release management. This is a critical view because it can remove a significant amount of source code from the set each product must subject to unit test as well as validation and verification efforts. This does not abdicate each product's responsibility for testing code developed for that specific product or for doing system level testing. However, it can significantly reduce the scope of testing.

4. IMPLEMENTING IN Ada

The Ada programming provides a number of features that naturally lend themselves to use when trying to implement the product line concepts. Although the notion of formal software product line architectures is newer than the Ada programming language, it is not surprising that the language would support their use. Ada was designed with the needs of developers of large systems with long lifecycles in mind such as avionics and ship control. These types of systems are often custom configured for customers and built in low volume production runs. This results in variations in hardware configurations that must be dealt with by system software [3]. Minimizing the impact of changing and evolving hardware configurations on the software asset base is critical to success. We have identified several features of the Ada programming language that can be used to implement a product line-based component architecture that can be adapted, extended, and tailored to meet the requirements of new applications.

4.1 Packages

Ada packages provide the most basic means to replace the implementation of a component in a product specific manner. Ada packages very effectively hide the inner workings of a package from outside users of a package. The inner workings of the package are contained in the package body that can be replaced without modifying the specification. In this way, the implementation of every subprogram within a package can be replaced. At this level, one ends up with copies of the source code for subprograms that are the same in the two bodies. Although this is certainly the most primitive method to provide an alternative implementation of a component, it is sometimes appropriate. For example, a component on the logical edge of the architecture often provides control of or interfaces to external devices or operating environments. When there are significant differences in these, there is often little in common between the multiple implementations. Consequently, providing completely independent bodies is a viable alternative.

4.2 Separate Subprograms

If it turns out that only certain capabilities vary between the configurations, then careful definition of subprograms and the use of Ada separates can be quite effective. Separate subprograms are often used to reduce recompilation because only the changed portions must be recompiled after

a change. But separates also permit easy incorporation of alternative subprograms.

For example, a reusable mathematical package may include alternate versions of an algorithm – one optimized for speed, the other for accuracy. When that package is configured for use by a particular application, the proper algorithm for that application can be configured.

The use of separate subprograms can also be used to hold subprograms that act as "configuration points." A simple example of a configuration point is one that returns a Boolean value indicating the presence of a specific feature. In a simple implementation, the function may return a hard-coded value while in a more complex implementation, it could check the state of a variable or switch set by the user. However, there are problems with using separate subprograms and their use should be minimized.

The Ada95 Style Guide [5] generally discourages the use of separate subprograms for a variety of reasons. It encourages embedding the subprogram into a package. One significant advantage of this approach is that having a package body provides a place for the subprogram to localize the data it requires. A cited disadvantage of separate subprograms is that one is a module and thus increases the number of separate modules in a system that must be tracked as part of maintenance.

4.3 Child Units

Child units can be a very effective means to encapsulate the behavioral differences between products. In contrast to separate subprograms, child packages can contain a set of related subprograms and any supporting data. Child units can be publicly visible or private to their parent. They are an effective means to

- extend the functionality of an existing package, or
- encapsulate a logical subsection of the package.

Being able to extend the functionality of an existing package is an attractive option when doing product line-based reuse. For example, consider a package providing support for a Global Positioning System (GPS) receiver with the parent *GPS* package containing only basic positioning capabilities. This reflects the most fundamental capabilities of all GPS receivers.

```
package GPS is …
```

However some GPS receivers have the additional capability of processing a waypoint list and this functionality could be placed in a public child package. Critically, this child package must be public in order for other packages in the system to access it. Also, clients of the child package *GPS.Waypoints* must explicitly "with" the child package.

```
package GPS.Waypoints is …
```

Note that the body of the child package *GPS.Waypoints* has visibility into the private part of *GPS*. In this way, it has great latitude in extending the capabilities of the parent package.

GPS receivers have a communications interface with the host computer that varies between specific GPS receivers. Although critical to the proper functioning of the *GPS* package, this interface is something that should be hidden from its external clients. Since the interface varies, it is also likely that this functionality will need to be modified or replaced to support a different GPS receiver. In this case, encapsulating this interface within a private child package segregates this functionality, thus making it easier to identify and replace.

```
private package GPS.Interface is ...
```

As a private package, *GPS.Interface* contains functionality only available to the package *GPS* and its other children. Specifically, it is **NOT** available for use by clients of the *GPS* package or its children.

Interestingly, both public and private child packages are applicable to the implementation of a product line-based reuse. By logically encapsulating functionality that is either optional or replaceable, child packages can be used to represent product variations.

4.4 Abstract Types and Subprograms

The Ada tagged type feature is used to extend data types. From a product line viewpoint, it is a way to augment an existing data type more information. Revisiting the GPS example, data required by the *GPS.Waypoint* child package could be added using tagged types as shown in the following example:

```
Package GPS is
  type GPS is tagged …
  procedure Display(G: GPS);
end;

with GPS; use GPS;
package GPS.Waypoints is
  type W is new GPS.Control with …
  procedure List;
end GPS.Waypoints;


package WaypointGPS is
  type WaypointGPS is new GPS with …
  -- new Display shows WayPoints
  procedure Display(G: WaypointGPS);
end;
```

Alternatively, one could use a more object-oriented approach and deriving a "GPS with Waypoint" as follows:

```
Package GPS is
  type GPS is tagged …
  procedure Display(G: GPS);
end;
with GPS; use GPS;

package WaypointGPS is
  type WaypointGPS is new GPS with …
  -- new Display shows WayPoints
  procedure Display(G: WaypointGPS);
end;
```

This is an alternative to using the parent/child package relationship that may be more appropriate in some circumstances. In the above case, the *Display* subprogram has been overloaded, although with upward conversion the subprogram *GPS.Display* is still accessible. Again this is a mechanism to reflect the product line view of the multiple applications within the software base.

An abstract type is a specific form of tagged type that is intended for use as a parent type for type extensions. Specifically, objects of an abstract type are not allowed. Similarly, abstract subprograms are subprograms that do not have a body and are intended to be overridden at some point when they are inherited. Packages with abstract types or abstract subprograms are intended to be the root classes of hierarchies.

Together, abstract types and subprograms allow the creation of hierarchies "is-a" or generalization-specialization relationships.

5. ORGANIZATIONAL ISSUES

No reuse effort can succeed on technical merits alone. For reuse of any kind to occur across individual product and product families, there must be commitment by the organization. Product family managers must recognize the benefits to sharing technology development and maintenance responsibilities across the individual products within the family they manage. Product family managers are also responsible for encouraging individual product teams to reuse existing components and develop new ones with reuse in mind. Similarly, corporate management must encourage reuse across product families for which they are responsible. At the highest levels of the organization, the goal should be to develop an intellectual property base that can be shared by all projects in the organization. Management should reward teams for reusing this intellectual property base and punish them when redundant development is knowingly done. Without management support, enterprise level reuse efforts are doomed to failure.

Management commitment is also necessary to establish a core architecture maintenance group that supports the organization's reuse base across multiple product families. This maintenance group must operate with the interests of all product families in mind. The core architecture of reusable components must be treated as a product itself even though the customers are strictly internal. The core architecture maintenance team should include representatives from each product family to ensure that all interests are considered. In addition, the organization should encourage communication between the core architecture group and product developers, as well as between developers on different product teams. Communication is essential to the long-term viability of the core architecture.

When developing a new product, components may be added to the reuse base in one of two different ways. The first case is when the development team realizes that an existing component from another product family can be reused in their application. The component must be analyzed to determine if it can be used with or without modification. If the component is used unmodified, then it can be added to the shared architecture. However, if the component requires modification, then it must be decided whether a second independent version of the component is to be created or whether a reusable version of the component can be created that is suitable for use by both projects. The analysis is strictly a technical issue. However, the decision on whether to create a second, independent version of an object or a shareable version is often driven by management rather than technical issues. If the development cost to make the component reusable is greater than that of making a second version, then a manager with short-term focus on cost will not consider the impact of distributing the cost of long-term maintenance. Similarly, if an organization does not reward contributions to the reuse base, then there is no obvious benefit to do so. Regardless of the final disposition of a particular component, it must be recognized by all involved that there are numerous technical and non-technical issues involved in the decision to make a component part of the reuse base.

The second case is when a new software component is developed for a specific product, and that new component is of direct use to other products. Without communication between the product groups, there is no way to know that the new component exists. Similarly, when a need arises, the product teams may know of an existing component that can be reused. If a component is of use to multiple product families, then it is a candidate for inclusion in the core architecture. Effort may be required to modify this component to make it suitable for reuse, but this effort is normally considerably less than implementing the component from scratch. However, since the modifications to make a component reusable are not of immediate direct benefit to a particular product, there is a tendency in many organizations to be resistant to investing this effort. Again, this ignores the fact, that if a component is reusable and merged into the core architecture, then maintenance becomes a shared expense and responsibility. Given the long lifespan of many products and the fact that

maintenance is a recurring cost, this alone should be justification enough to aim for reusability whenever possible. As the maintenance of the reusable software components is not tied to a particular product but instead spread across multiple products, the overall software maintenance costs of the organization are reduced.

In a sense, the goal of management and the core architecture team is to foster a sense of community and joint ownership. It is in the best interest of all parties to support the core architecture and promote its user. Projects that successfully use the open source development model foster this sense of joint ownership and community via newsgroups, mailing lists, and web sites. This sense of *togetherness* is critical to the survival of the enterprise as a whole. It is not enough that individual products succeed; the organization's entire line of products should succeed.

It is critical to have someone who understands each product family and acts as system architect for the product family as a whole. This ensures that there is compatibility and consistency across the product line. This person is also key to planning product family upgrades and feature enhancements. Without someone acting as system architect, there is no single person who understands how the individual products within a product family relate to and complement one another [2].

Similarly, there should be a system architect for the core architecture. This person ensures the components in the core architecture are of general use to multiple product families. Otherwise, the core architecture can become a dumping ground for software components that projects do not want to maintain on their own.

Another important role is that of reuse champion and technology broker [4]. If the sense of community is fostered, everyone is to some extent a reuse champion. Reuse champions encourage the reuse of existing components and development of new components with reuse in mind. Ideally, individuals are excited when software is used by multiple projects and get a sense of personal gratification when other products use their work. The role of the technology broker is subtly different. The technology broker is involved with the design of new products within existing product families and new product families. The technology broker's goal is to help recognize when an already existing component can be reused and when a new component offers the potential for reuse. The technology broker's goal is to reduce the development effort required for new projects. Moreover, when new development of a reusable component is required, the broker guides in the design and implementation of that component to ensure that it can be added to the enterprise asset base with minimal effort.

The bottom line is that the individuals in the organization must actively support the idea of software reuse across product families. This applies equally to the most junior members of the development staff, as well as the most senior members of management. Moreover, encouraging projects that could reuse the core architecture but do not use it is detrimental to the overall success of the reuse effort for both technical and morale issues.

6. BENEFITS

Software reuse brings many benefits both at the individual project and enterprise levels. Key among these are increased reliability, reduced time to market, reduced development cost, and reduced software maintenance cost. A product line-based approach enhances reuse efforts by elevating the developers' reuse view above simple data structures and algorithms to include high level product capabilities.

Increased reliability is a benefit of effective reuse because applications are no longer composed of completely new software. They include reused components that are mature and fully tested. Problems in the implementation of the component have been discovered and resolved by previous users of the component. As time progresses, the number of products using a component increase and the probability of uncovering a new defect in the component diminishes.

Reduced time to market occurs because the time required to design, develop, and test an application is reduced. A software reuse base provides developers with software components that can be included in a new application with no development time. Moreover, the core and product family architectures are toolkits that provide a baseline for starting new application development. The core architecture provides a base line for application development regardless of the intended product. It can be used with little investment of effort by the developer of a new product family. This core architecture will likely include development tools, operating systems, device drivers, and an application framework. This provides a robust starting point for new product families.

The product family architecture takes the core architecture to the next logical level. It includes all the functionality shared across products within a family and can be used as the basis for new product variants without redeveloping or assembling existing components as a starting point. The goal of the product family architecture is to make it possible for a developer to immediately begin to develop the new functionality that makes their new variant unique. Common functionality is already included and easily shared. The ease of reusing existing software components with other product family members ensures compatibility between product family members. In short, a software reuse base not only provides a launchpad for developers to initiate new products from, but also leverages existing software components so developers can focus on the functionality that makes their product family or product variant unique.

Just as using a core architecture and product family architecture reduced the length of time required to develop

new products and bring them to market, it also **reduces the development cost**. By focusing on reusing and sharing software components, there is an emphasis on eliminating the development of existing functionality. This eliminates the unnecessary cost of multiple products developing and testing the same functionality. Developers are encouraged to think in terms of multiple products and can sometimes **anticipate future product options** and plan for them.

In addition, when a development environment (host computer and tools) are included in the core or product family architectures, then new products and product families will not be required to repeat tool evaluation efforts. Moreover, there will be an enterprise knowledge base built for those tools, thus increasing the efficiency of developers using and system administrators installing them.

Software maintenance costs are reduced because core and product family architectures centralize the maintenance of the reuse base. The common practice of copying code from an existing application and including that source in a new application avoids the cost of developing the software component from scratch, but forces each project with a copy of a component to independently maintain it. When an improvement or bug fix is made to any one of these copies, there is no mechanism in place to ensure that the modification is propagated to all users and properly tested. The same modification could independently and redundantly be made to each copy. Even worse, those modifications could be made in different ways and with varying quality assurance controls. At this point, there is no way to know which of the many independent versions should be copied next time. And each individual product is responsible for integrating the components it selects for reuse.

Effective reuse requires that there be a single “official” source for the reusable components. All maintenance of the component is done at this single point and tested releases of the collected components are made to ensure the interoperability of components within the reuse base. In this way, the overall quality of the reuse base is increased while eliminating the problem that each user of the component has an independently maintained copy. Multiple products share the cost of maintaining that reusable component. This distributes maintenance costs across all users of the reuse base. The cost to each product for maintaining that software component is a fraction of what it would be if it were not reused.

In many organizations, all products share equally in the maintenance of the product family and core architecture components that they may reuse. However, this is not a requirement of this reuse model. There are legitimate cases in which a single product family should incur a greater share of the cost of the core architecture. For example, a single product may want to port the core architecture to a system that is of interest to no other product or product family. This could occur when a product manager wishes to support an obscure, possibly obsolete, hardware platform that no competing vendor is willing to support. This would provide a captive market for the product. In this case, the cost of porting the core architecture to this obscure hardware platform and maintaining it on that platform should solely be the responsibility of a single product.

Ultimately, the distribution of maintenance costs of the core and product family architectures should be shared in a fair and equitable way by all products using them. When costs are distributed unfairly, this can work to undermine the entire reuse effort.

7. APPLICATION TO EXISTING PRODUCTS

It is seldom that families of applications are designed from the onset to share components with one another. The more common scenario is that an application is successfully deployed without consideration of reuse issues. Subsequently, a copy of the entire application source code is made so that modifications can be made to satisfy special user communities, to target a new user base, or to create a foreign language version. This creates a second version of the application that may share 95% or more of its source code with the original version. Unfortunately, these variants of the original application exist independently and are maintained independently. Since there is no formal reuse in place, bug fixes are made to one version and not the other. Changes may be made to the user interface of one version and not to the other, thus making the two versions appear different to an end user. File formats may be changed without regard for compatibility with other application variants. At this point, someone realizes that maintenance would be much simpler and cheaper if the two versions of the application could share maintenance of the common source code.

After the decision has been made to create a shared code base between the two application variants, the next step is to analyze the source code of the application variants. The source code must be compared to determine which software components are the same and which are different. Those that are the same can automatically become part of the shared code base. Those that are different must be analyzed to determine why there is a difference. If the difference can be resolved in a way that both applications can share the same version, then the component in question may be added to the shared code base. Some software components may require rework to become reusable. At the end of the analysis, all of the application software components will have been categorized as either specific to the original application, the application variant, or shared between the original application and the variant.

Once a shared code base has been created, each application must be recompiled from source code to take advantage of the new shared code base. After this recompilation, each application must be retested to ensure that using a shared

code base does not negatively impact the behavior of the application. After both application versions have been requalified, the maintenance of the shared code base becomes the shared responsibility of the application groups. Previously, both application groups were required to independently maintain copies of the shared code. This reduces the overall maintenance cost of those components by fifty percent.

Moving to a shared code base once is not the end of this process. There must be a commitment on the part of both developers and management to continue to use the shared code base. If a third application variant is created by simply copying the shared code base with no plan to reuse it, then the effort to create the shared code base is wasted. This ongoing commitment to reuse is the difference between success and failure. Once an organization decides to share code between various products, it must commit to ensuring that new products will reuse the shared code base. Otherwise, the full benefits of reuse cannot be realized.

8. CONCLUSION

Software reuse based on product families is an effective way to leverage product similarities to reduce both development and maintenance costs. It requires effort and commitment by both technical and management personnel to establish a reuse group. But this investment is rewarded as more products are based on the core architecture. Many may consider it a waste to spend extra time and money submitting modifications for reuse or consulting with other projects in order to keep shared functionality. This negative view is unfortunately common as deadlines loom and pressures to ship mount at the end of a project. If someone views ***product release*** as the end of a product's life, then it will be all but impossible to encourage that person to contribute to a reuse base.

Maintenance is a fact of life -- there will be product improvements, technology insertions, bug fixes, and other changes after the product initially ships. By establishing and effectively utilizing a reusable code base, an organization will realize numerous benefits including lower maintenance costs both at an individual product and organization level, increased reliability, and reduced time to market. Importantly, there are other, less obvious, benefits such as migration of bug fixes across products and promotion of a common look and feel across products.

Reuse is not a one-time event, but a continuous process based on a commitment to the long-term success of the organization. Just as any relationship requires commitment, shared goals, and attention to detail, so does a successful reusable code architecture.

9. REFERENCES

[1] International Standard ISO/IEC 8652:1995(E) Ada Language Reference Manual. International Organization for Standards/International Electrotechnical Commission. 1995.

[2] Berghel, H. The Cost of Having Analog Executives in a Digital World. Communications of the ACM. (November 1999) 11-13.

[3] Clements, P. Software Product Lines: A New Paradigm for a New Century. CrossTalk: The Journal of Defense Software Engineering (February 1999) 20-22.

[4] Dufaud, S.B. More Experience with a Technology Transition Broker. CrossTalk: The Journal of Defense Software Engineering (December 1994).

[5] Software Productivity Consortium. ADA 95 Quality and Style: Guidelines For Professional Programmers. SPC-94093-CMC. Version 01.00.10. October 1995. Available online at http://www.adaic.org/docs/style-guide/95style/.