

Reuse through Software Product Families

Magne Syrstad

25 April 2003

1 Introduction

Software product families is a concept created to help companies reuse work products between similar applications. They are inspired by the ideas previously used by people like Henry Ford in automobile production. The main idea is to emphasize proactive reuse, interchangeable components, and multi-product planning cycles in order to construct high quality products faster and cheaper [6].

A discussion of the ideas behind software product families is the main part of this essay. Before moving over to them, there will be a shorter introduction to reuse in the general case, and at the very end there will be a short conclusion.

When reading about software product families, one will often find the term *software product lines* used. These describe the same concept, the first term being of European origin, the second of American [9].

2 Software Reuse

Software reuse has been going on, in more or less organized form, for a very long time. One of the simplest forms of reuse is using cut and paste of code-lines in order to use a particular piece of code more than once. More advanced forms of reuse includes the use of tools and the reuse of functions.

Today, the goal is often considered to be to develop software by putting together COTS or reusable, internally developed, complex components, and only customizing the code that glues these components together.

The usual aim of all reuse efforts, at all levels, is to save time and effort in order to be able to deliver software at lower cost, as well as lowering the time to market.

2.1 Cost-Effective Reuse

In order to make cost-effective reuse, one has to follow simple economic principles. Barnes and Bollinger [1] shows that reuse should be treated as an investment. Reuse has an initial cost - the investment made. For each activity using the reused work product, one has to consider the savings obtained by the reuse. If the total savings obtained by this reuse, for all the reuse cases, is greater than the investment, the particular instance of reuse is cost effective.

What follows from this line of thinking, is three different ways of making reuse more profitable. First, one can reduce the investment cost, that is, lowering the initial investment. Second, one can increase the number of reuses. Third, one can reduce the cost of reuse in itself, which is the cost of integrating the reusable asset.

2.2 Which Work Products to Reuse

In the above, the impression is given that pieces of code is the only thing to be reused in the creation of a software system. Barnes and Bollinger [1] takes the view that *human problem solving* is a scarce resource, and that reuse, automation (like code generation and compilers) and good planning are the methods available to reduce the need for human problem solving.

Focusing on reuse, they point out that in order to reuse human problem solving, one has to aim for *broad-spectrum reuse*. This includes reusing work products like requirements specifications, specifications, designs and documentation in addition to code modules. They further claim that reusing the early work products is very important, as they by themselves require substantial use of human problem solving. In addition to this, the reuse of early work products greatly increases the likelihood of reuse of the work products earlier developed on the basis of these.

Also, the trivial reuse by cut & paste are not only code reuse, there is at the same time an informal reuse of the designs and the ideas behind those particular lines of code.

2.3 Making Reuse Work

Reuse may sound simple, but experience has shown that reaching more than casual reuse through cut & paste and simple tools requires particular care when it comes to organization. Fafchamps [4] gives an overview of 4 different organizational methods used in reuse programs at Hewlett-Packard,

including their advantages and disadvantages. The focus of the different organizational patterns are how the producers (those that create the reusable parts) are placed in the organization compared to the different teams of consumers (that makes use of the reusable parts in different products).

Poulin [8] describes the experiences from “formal” reuse efforts at IBM. A reuse repository was established, but they soon realized that what was put into the library, was mostly low-level utilities and functions. The typical situation is that almost all reuse between organizations comes from a set of domain-independent software that does not exceed 15 to 20% of the total product. Of all reported reuse, the largest part (up to 85%) comes from domain-specific software created by domain experts.

This is similar to the typical reuse library. There may be a lot of components added to the library, but very few of them are used, and those used are mostly small, general-purpose components. Poulin claims a typical large corporate reuse library has the progress from starting with very few parts, to become a library of many parts of low or poor quality, until it ends up as a library of many parts of little or no use. He further shares experiences from local incentive programs that reward those that supply reusable software that meets a specific standard, as well rewarding those that reuse software in their projects.

At IBM, they experienced that whatever the efforts to make a company-wide reuse-program successful, this failed due to the very limited amount of software that has a broad enough scope. Poulin makes the conclusion that while some domain independent software may be successful in a reuse context, this only makes up a small part of the total applications. Domain specific software, on the other hand, is a better candidate for reuse, but then of course, only within the domain it is made for.

3 Software Product Families

As mentioned in the introduction, software product families are based on the idea of creating software from interchangeable components, through proactive reuse and multi-product planning cycles. The term software product family, implies a particular mind-set. Having multiple products in a domain does not make them a software product family unless the ideas above are implemented [7].

Creating a software product family is not a trivial task. Adapting to this approach has challenges depending on whether the products already exists, or you are faced with starting from scratch. There is an initial investment

to be made to prepare the infrastructure needed. The foundations of the infrastructure must be well thought through, as they will be difficult to change afterwards. There are also managerial challenges in how to organize a company between producing the reusable assets and creating the products based on these. The sections below will give more details about these challenges.

3.1 Essential Activities

Northrop [7] gives an overview of the essential activities in creating and administrating a software product family. The essential activities are:

Core asset development This activity delivers the product scope, the core assets and the production plan. The core assets, sometimes referred to as the product family platform, are the basis for the product family. These include the product family architecture, the reusable software components, domain models, documentation, etc. The core assets will be further described in section 3.2. The inputs to this activity are among others the inventory of pre-existing assets, the production strategy, existing styles, patterns, frameworks and constraints on the production (internal or external).

Product development The output from this activity, is the finished products. The inputs of this activity are the requirements for the individual products, the scope of the product family, the core assets and the production plan. The production of the final products can have a strong feedback on the scope and core assets of the product family.

Management Management is divided into technical and organizational management. Technical management oversees the development of both core assets and products. Organizational management is responsible to create the proper organization to make the production process efficient. In addition to this, organizational management is responsible for giving the core asset and production development teams sufficient resources when it comes to staff.

It is important to remember that the essential activities goes on at the same time, and that they influence each other. For instance, a change in the product scope (a core asset) may lead to the creation of a new product (a product development activity), which again requires a change in the organizational structure of the organization in question (a management activity).

3.2 Fundamental Work Products

This section will describe the core asset of the software architecture and the scope. McGregor et al [6] points out that the software architecture is the key to the success of a software product family. One reason for this is that the software architecture, along with the shared components, has a great influence of which products that will easily be created inside the software product family. Also, there should be taken great care in designing the shared components, as they may be expected to evolve over time. Clements [3] on the other hand, points at the scope as the key to position the product family properly in the market. The sections below will give an overview of both.

In section 2.2 it is pointed out that it is most effective to reuse are the early work products. Software product families focuses on establishing a common platform and a well defined scope. In this they do focus on reusing the first work products, as well as complete and advanced components. Software product families also follows the conclusion of Poulin [8] in that reusable components have to be domain-specific. By nature, the scope of a product family places the products in the same domain.

3.2.1 The Software Architecture

As mentioned elsewhere, the software architecture is by many regarded as the most important asset in a software product family. Northrop points out: “The lack of either an architecture focus or architecture talent can kill an otherwise promising product line effort” [7].

In many ways, defining a product family architecture is similar to defining any system architecture. The main difference is that a product family architecture has to allow for many products to be based on it, so it must allow for variability. Bosch [2] points out two extremes on how to allow for variability. One extreme is that the software architecture is used as-is by all products. This would require the difference between the products to be implemented through the variability in each of the components. Another extreme is that the product line architecture is used only as a core or a part of the product architectures, where the product architecture must extend the architecture into a complete product.

In addition to allowing for variability, a product family architecture has to take into consideration the requirements for all planned products. This is particularly important to consider when it comes to non-functional requirements, that is requirements for the quality attributes for the system.

Changing an existing system when it comes to quality attributes is in the typical case very difficult. The variability mentioned about can be seen as a quality requirement when it comes to modifiability and portability.

The basis for an architecture can be frameworks like J2EE or .Net. Building blocks for an architecture includes things like architectural styles and patterns. Changing the patterns used can be the basis for transforming an architecture, as patterns typically influences quality attributes. Other variation techniques includes optionality of certain components. Bosch gives a total of four types of architecture transformation [2].

The architecture typically decides the main components and connections of the system. An important decision to make is whether to use a COTS component, or develop components internally. This decision typically depends on the availability and price of available COTS components, as well as the resources and expertise available locally for developing the components in question.

3.2.2 The Scope

The scope of the software product family can be viewed as a doughnut across all possible products [3]. Those products that are inside the doughnut are all systems that the organization is preparing to build within their product family (even if the actual production of them does not have to be decided from the start). Products that are on the doughnut itself, are products that could be built, but require extra effort and consideration.

Carefully representing the scope, allows for examining the border-regions for areas that could be fitted into the product line, and that is underrepresented by actual products. Having such product opportunities at the “edge” of the scope of the product line, allows for quickly creating a product filling the gap. A proactively decided scope, helps companies plan their own future and increase their agility when it comes to capturing new market segments [3].

Bosch [2] points out that what is inside the scope is cheap to develop. However, if the scope is large, the shared components will have to be very complex, including functionality that a lot of products won’t need. Making functionality optional may be expensive and difficult in practice, leading to products being unnecessary large when it comes to resource usage. Another alternative to disabling features, is to have multiple versions of the same component of different feature-richness. This, of course also has its drawbacks in leading to more parts to maintain.

A method for deciding the scope of a product family is given by Bosch [2].

3.3 Adopting a Product Family Approach

There are several ways of adopting a product family approach. Krueger [5] points out three predominant adoption models. These are:

Proactive This approach is similar to the waterfall model. All foreseeable products are analyzed, designed and implemented up front. This requires that requirements can be predicted well into the future, and that one has the time and resources to put in a lot of work up front.

Reactive Reactive adoption on the other hand, is more like the approaches found in extreme programming. Analyzing, designing and implementing goes on in incremental steps in a development spiral. This method is suitable where one can not see the complete set of requirements up front, and in situations where there is no time to take a pause in the production of new version or there is little extra resources available for establishing the product family.

Extractive This is an approach that bases a product family on a set of existing products. Existing software is reused, and adapted without too much re-engineering. This method is well suited to quickly move a traditional product portfolio into a software product family.

Bosch [2] speaks of evolutionary and revolutionary methods. Evolutionary methods are similar to the reactive and extractive approaches described above, while revolutionary methods are similar to the proactive approach. Other terms used for the same or similar concepts are lightweight and heavy-weight approaches [6]. The properties of evolutionary and revolutionary approaches adapted from Bosch are discussed below:

3.3.1 Evolutionary Methods

If there is an existing set of products to be evolved into a product family, the first step to take is to decide a product family architecture based on the architecture of the existing products. Then, components that fills the requirements for more than one product must be identified, and starting with the most important, a product component must be generalized to a product family component. Each time a product family component is finished, the products that are to be a member of the product family must be adapted to use the new shared component rather than their previous individual one. By generalizing more and more components, the company gradually moves from a traditional product based approach to a software product family.

If an evolutionary method is to be used with a new software product line, the shared components are gradually evolved and expanded as more products are added. This reduces the risk when moving into a new domain, as the requirements are not as well understood as in the case where one already has an existing set of products. However, adapting the shared components to fit new products may become very costly if their architecture is not suited for making the changes that is about to take place.

In general, evolutionary methods reduces the initial cost of implementing a software product family. Through the lower initial cost, the risk involved in establishment of the product family is reduced. However, it may not be the best solution in the long term, if many products are to be added, as the cost of making the shared components fit the new product may become high.

3.3.2 Revolutionary Methods

Revolutionary methods, or proactive methods also plays out a bit different depending on whether there is a set of products to be replaced or not. In the case where an existing set of products does exist, the preferable way of progressing is to create a brand new product family, without using the old products for other than reusing requirements and experiences. The problem with this, is that there will be a pause in the release of new products while the new product family is being established. This pause may be too long for many companies, who would be better served by an evolutionary approach that keeps the production going as the product line is established.

If there is no previous product to replace, there will have to be taken particular care in finding all requirements that the future products will have. The shared components would have to be based on a super-set of the requirements identified for each individual product.

The advantages of revolutionary methods is that the total cost of establishing a product family usually is smaller compared to evolutionary methods. When the product family is established, the products can be developed very rapidly. The disadvantages are mostly concerned with time and risk. The establishment of a brand new product family stops or greatly reduces the evolution of the existing products. This may lead to a loss of income during the conversion. In addition to this, the great investment that has to be done up front, could lead to a financial disaster if the adoption process fails. Compared to the evolutionary methods, this revolutionary, or proactive approach ha the trait of high risk and high gain.

3.4 Organization

An organization running a product family has two concerns. One is to develop and maintain the core assets that are common between all products, the other is to develop and maintain each of the final products. McGregor et al [6] claims that a two-tiered organization is a key to success, but does not go into detail of how the two concerns are to be distributed. Bosch [2] gives 4 different organizational models for a product family oriented organization. These are:

Development department All software development is concentrated in a single development department. All members in this department can in principle be assigned to work on all product or the core assets as seen fit. Staff is dynamically assigned to the tasks where they are most needed. This organizational form is typically applicable in relatively small organizations, with less than approximately 30 staff members. The advantages of this organizational form is that there is very good communication between people working on different parts of the product line. The main disadvantage is that it does not scale well to larger organizations. Another disadvantage is that the flexible organization may lead to less popular tasks being taken lightly.

Business units The organization is divided into business units. Each business unit typically has the responsibility for one product. The common assets are shared among all the business units, with each unit making the changes they need. Bosch has observed three major variants of this organizational form.

In an *unconstrained model* any business unit can make any change to any shared asset. This can lead to the problem of the shared components having too product specific functionality, which again leads to degradation of the component over time.

In a model with *asset responsables* one software engineer is assigned as responsible for each common asset. The task of the responsible engineer is to make sure that the asset is evolved according to the best interest to the organization as a whole rather than according to the interest of a single business unit. The business units still perform their own changes to the common assets, but only after acceptance by the engineer responsible. In theory, this should solve the problems of degradation of quality, but in practice time constraints often makes it hard to avoid adding product specific parts when time is short.

Mixed responsibility is a model where the each business unit gets the responsibility of a shared asset in addition to their primary product. The responsible unit should preferably be the one that makes the most advanced use of the asset, so that most change requests comes from within the business unit, reducing the communication overhead. This gives better control of each shared asset, as other business units are no longer allowed to make changes to the asset. A problem may be that communication is more difficult in this case if a unit must ask a different business unit to make the change, which could lead to delay in the completion of products.

Domain engineering unit The work on the shared components are given to a domain engineering unit. The products are made by multiple product engineering units. The domain engineering unit is responsible for all development of the shared components. While the product development units have contact with the customers of their specific products, the domain engineering unit has no such direct contact, and relies on the product engineering units to give them the information they require in order to move the common assets in the right direction.

This organizational form is mostly applicable in quite large organizations. If the organization is very large, it may be beneficial to make multiple domain engineering units that distribute the responsibility for the core assets among them. Smaller organizations are usually negative with regards to this organizational form, fearing that the domain engineering unit will be too fixed on making the “perfect” solution, rather than delivering products.

The advantages of this model, is that there is no need for complex communications between the product engineering units, as may be the case in the previously presented alternatives. Also, conflicts between product teams when it comes to which features to implement in the shared components, can be more objectively decided when an external unit is responsible for the asset. Also this organizational form scales better to large organizations than the previously mentioned models.

This model also has its disadvantages. The flow of requirements to the domain engineering is one of them. While the domain engineering unit delivers reusable components to the product units, the requirements has to flow in the opposite directions. The need to balance the requirements from different directions, may lead to an increased time to marked of products. This may be a major disadvantage, as shorter time to marked is one of the prime benefits of a product family approach. This can be overcome by allowing product teams to create temporary versions of the shared assets, but this

can again lead to other problems.

Hierarchical domain engineering units This is the last organizational model described by Bosch [2]. In this situation there is, as the name implies, a hierarchy of domain engineering units. This model is suited for very large organizations, with hundreds of domain engineers, or in situations where the number of products is very large. There is a significant communication overhead with this model. Bosch gives the mobile phone division of Nokia as an example of a company using this organizational structure.

4 Conclusion

This essay has introduced software product families. Software product families is a concept for reuse through a very systematic and proactive effort. The products produced in the product family are based on the same platform. The importance of a well thought through software architecture and a well defined scope has been emphasized.

Getting a software product family up and running allows for cheaper product development and shorter time to market. However, there are pitfalls when designing a software product family, and failures at the creation of the product family can be very expensive. There are several success-stories associated with software product families [7], but there are of course also histories of failures.

References

- [1] Bruce H. Barnes and Terry B. Bollinger. Making reuse cost-effective. *IEEE Software*, pages 13–24, Jan. 1991.
- [2] Jan Bosch. *Design @ Use of Software Architectures: Adopting and evolving a product-line approach*. Addison Wesley, Harlow, UK, 2000.
- [3] Paul Clements. Being proactive pays off. *IEEE Software*, 19(4):28–31, Jul./Aug. 2002.
- [4] Danielle Fafchamps. Organizational factors and reuse. *IEEE Software*, pages 31–41, Sep. 1994.
- [5] Charles Krueger. Eliminating the adoption barrier. *IEEE Software*, 19(4):29–31, Jul./Aug. 2002.

- [6] John D. McGregor, Linda M. Northrop, Salah Jarrad, and Klaus Pohl. Initiating software product lines. *IEEE Software*, 19(4):24–27, Jul./Aug. 2002.
- [7] Linda M. Northrop. SEI’s software product line tenets. *IEEE Software*, 19(4):32–40, Jul./Aug. 2002.
- [8] Jeffrey S. Poulin. Populating software repositories: Incentives and domain-specific software. *Journal of Systems and Software*, 30:187–199, 1995.
- [9] Frank van der Linden. Software product families in europe: The esaps and café projects. *IEEE Software*, 19(4):41–49, Jul./Aug. 2002.