

EXPANDING THE SCOPE OF SOFTWARE PRODUCT FAMILIES: PROBLEMS AND ALTERNATIVE APPROACHES

Jan Bosch
Nokia Research Center
P.O. Box 407, FI-00045 NOKIA GROUP, Finland
Jan.Bosch@Nokia.com, <http://www.janbosch.com>

ABSTRACT

Software product families have found broad adoption in the embedded systems industry. Product family thinking has been prevalent in this context for mechanics and hardware and adopting the same for software has been viewed as a logical approach. During recent years, however, the trends of convergence, end-to-end solutions, shortened innovation and R&D cycles and differentiation through software engineering capabilities have led to a development where organizations are stretching the scope of their product families far beyond the initial design. Failing to adjust the product family approach, including the architectural and process dimensions when the business strategy is changing is leading to several challenging problems that can be viewed as symptoms of this approach. This paper discusses the key symptoms, the underlying causes for these symptoms as well as solutions for realigning the product family approach with the business strategy.

1. INTRODUCTION

Mobile phones have, over the last decade, evolved from basic devices aimed primarily at the voice and SMS use cases to a rich set of mobile devices ranging from mobile multi-media computers to mobile enterprise devices. Contemporary mobile devices support a rich set of use cases including taking still and video pictures, playing music, watching television, reading email, instant messaging, navigating, etc. In response to the enormous increase in the features required from mobile devices, the demands on the software present in the mobile device have increased similarly. The developments in the mobile devices industry are illustrative examples of the general trend in embedded systems: the investment in software R&D has increased by an order of magnitude during the last decade.

One can identify three main trends that are driving the embedded systems industry, i.e. convergence, end-to-end functionality and software engineering capability. The convergence of the consumer electronics, telecom and IT industries has been discussed for over a decade. Although many may wonder whether and when it will happen, the fact is that the convergence is taking place constantly. Different from what the name may suggest, though, convergence in fact leads to a portfolio of increasingly diverging devices. For instance, in the mobile telecom industry, mobile phones have diverged into still picture camera models, video camera models, music player models, mobile TV models, mobile email models, etc. This trend results in a significant pressure on software product families as the amount of variation to be supported by the platform in terms of price points, form factors and feature sets is significantly beyond the requirements just a few years ago. The second trend is that many innovations that have proven their success in the market place require the creation of an end-to-end solution and possibly even the creation or adaptation of a business eco-system. Examples from the mobile domain include, for instance, ring tones, but the ecosystem initiated by Apple around digital music is exemplary in this context. The consequence for most companies is that where earlier, they were able to drive innovations independently to the market, the current mode requires significant partnering and orchestration for innovations to be successful. The third main trend is that a company's ability to engineer software is rapidly becoming a key competitive differentiator. The two main developments underlying this trend are efficiency and responsiveness. With the constant increase in software demands, the cost of software R&D is becoming unacceptable from a business perspective. Thus, some factor difference in productivity is easily turning into being able or not being able to deliver certain feature sets. Responsiveness is growing in importance because innovation cycles are moving increasingly fast

and customers are expecting constant improvements in the available functionality. Web 2.0 [O'Reilly 05] presents a strong example of this trend. A further consequence for embedded systems is that, in the foreseeable future, the hardware and software innovation cycles will, at least in part, be decoupled, significantly increasing demands for post-deployment distribution of software.

Due to the convergence trend, the number of different embedded products that a manufacturer aims to bring to market is increasing. Consequently, reuse of software (as well as of mechanical and hardware solutions) is a standing ambition for the industry. The typical approach employed in the embedded systems industry is to build a platform that implements the functionality common to all devices. The platform is subsequently used as a basis when creating new product and functionality specific to the product is built on top of the platform.

Although the platform model is easy to understand in theory, in practice there are significant challenges. As discussed in [Bosch 00], the platform model is supposed to capture the most generic and consequently the least differentiating functionality. Innovations and differentiating functionality are supposed, over time, to flow from product-specific implementations to the platform. However, in practice one can identify at least two forces that drive innovations directly to the platform. First, in several cases, it is clear that a novel feature or innovation will be required by all or most future products. In this case, there is a clear rationale to implement the new feature or innovation directly in the platform, bypassing the product-specific phase. Second, in the case that a company also licenses the platform to other organizations in the industry, the platform itself needs to be differentiating and contain sufficient novel features to hold or expand its position in the face of competition from other platforms.

Second, the product specific functionality frequently does not respect the boundary between the platform and the software on top of it. Innovations in embedded systems can originate from mechanics, hardware or software. Both mechanical and hardware innovations typically have an impact on the software stack. However, due to the fact that the interface to hardware is placed in device drivers at the very bottom of the stack and the affected applications and their user interface are located at the very top of the stack, changes to mechanics and hardware typically have a cross-cutting effect that causes changes in many places both below and above the platform boundary. A second source of cross-cutting changes is software specific. New products often enable new use cases that put new demands on the software that can not be captured in a single component or application, but rather have architectural impact. Examples include adding security, a more advanced user interface framework or a web-services framework. Such demands result in cross-cutting changes that affect many places in the software, again both above and below the platform boundary.

Software product families have, in many cases, been very successful for the companies that have applied them. Due to their success, however, during recent years one can identify a development where companies are stretching their product families significantly beyond their initial scope. This occurs either because the company desires to ship a broader range of products due to, among others, convergence, or because the proven success of the product family causes earlier unrelated products to be placed under the same family. This easily causes a situation where the software product family becomes a victim of its own success. With the increasing scope and diversity of the products that are to be supported, the original integration-oriented platform approach increasingly results in several serious problems in the technical, process, organizational and, consequently, the business dimension.

The purpose and contribution of this paper is that it analyses the aforementioned problems and to present alternative approaches that are better suited for broad-scoped product families. Both the problem analysis and the proposed alternative approaches are based on experiences from a variety of companies that the author has worked with during the last decade. However, for reasons of confidentiality, no specific references can be provided at this point.

In the remainder of this article, we first present a more detailed assessment of the problems and challenges associated with the traditional, platform-based, integration oriented approach. Subsequently, we discuss five aspects of software product families that are most relevant when broadening the scope of a product family in section 3. We then proceed with presenting two alternative approaches, i.e. hierarchical product families and the compositional-oriented approach, in section 4. Finally, related work is described in section 5 followed by the conclusions of the paper.

2. INTEGRATION-ORIENTED APPROACH: PROBLEM STATEMENT

This paper discusses and presents the concerns of the integration-oriented platform approach. However, before we can discuss this, we need to first define integration-oriented platform approach more precisely. In most cases, the platform approach is organized using a strict separation between the platform organization and the product organizations. The platform organization has typically a periodic release cycle where the complete platform is released in a fully integrated and tested fashion. The product organizations use the platform as a basis for creating and evolving their product by extending the platform with product-specific features.

The platform organization is divided in a number of teams, in the best case mirroring the architecture of the platform. Each team develops and evolves the component (or set of related components) that it is responsible for and delivers the result for integration in the platform. Although many organizations have moved to applying a continuous integration process where components are constantly integrated during development, in practice significant verification and validation work is performed in the period before the release of the platform and many critical errors are only found in that stage.

The platform organization delivers the platform as a large, integrated and tested software system with an API that can be used by the product teams to derive their products from. As platforms bring together a large collection of features and qualities, the release frequency of the platform is often relatively low compared to the frequency of product programs. Consequently, the platform organization often is under significant pressure to deliver as many new features and qualities during the release. Hence, there is a tendency to short-cut processes, especially quality assurance processes. Especially during the period leading up to a major platform release, all validation and verification is often transferred to the integration team. As the components lose quality and integration team is confronted with both integration problems and component-level problems, in the worst case an interesting cycle appears where errors are identified by testing staff that has no understanding of the system architecture and can consequently only identify symptoms, component teams receive error reports that turn out to originate from other parts in the system and the integration team has to manage highly conflicting messages from the testing and development staff, leading to new error reports, new versions of components that do not solve problems, etc.

Although several software engineering challenges associated with software platforms have been outlined, the approach often proves highly successful in terms of maximizing R&D efficiency and cost-effectively offering a rich product portfolio. Thus, in its initial scope, the integration-oriented platform approach has often proven itself as a success. However, the success can easily turn into a failure when the organization decides to build on the success of the initial software platform and significantly broadens the scope of the product family. The broadening of the scope can be the result of the company deciding to bring more existing product categories under the platform umbrella or because it decides to diversify its product portfolio as the cost of creating new products has decreased considerably. At this stage, we have identified in a number of companies that broadening the scope of the software product family without adjusting the mode of operation quite fundamentally leads to a number of key concerns and problems that are logical and unavoidable. However, because of the earlier success that the organization has experienced, the problems are insufficiently identified as fundamental, but rather as execution challenges, and fundamental changes to the mode of operation are not made until the company experiences significant financial consequences.

The problems and their underlying causes that one may observe when the scope of a product family is broadened considerably over time include, among others, those described below:

- **Decreasing complete commonality:** Before broadening the scope of the product family, the platform formed the common core of product functionality. However, with the increasing scope, the products are increasingly diverse in their requirements and amount of functionality that is required for all products is decreasing, in either absolute or relative terms. Consequently, the (relative) number of components that is shared by all products is decreasing, reducing the relevance of the common platform.
- **Increasing partial commonality:** Functionality that is shared by some or many products, though not by all, is increasingly significantly with the increasing scope. Consequently, the (relative) number of components that is shared by some or most products is increasing. The typical approach to this model is the adoption of hierarchical product families. In this case, business groups or teams responsible for certain product categories build a platform on top of the company wide platform. Although this alleviates part of the problem, it does not provide an effective mechanism to share components between business groups or teams developing products in different product categories.
- **Over-engineered architecture:** With the increasing scope of the product family, the set of business and technical qualities that needs to be supported by the common platform is broadening as well. Although no product needs support for all qualities, the architecture of the platform is required to do so and, consequently, needs to be over-engineered to satisfy the needs of all products and product categories.
- **Cross-cutting features:** Especially in embedded systems, new features frequently fail to respect the boundaries of the platform. Whereas the typical approach is that differentiating features are implemented in the product (category) specific code, often these features require changes in the common components as well. Depending on the domain in which the organization develops products, the notion of a platform capturing the common functionality between all products may easily turn into an illusion as the scope of the product family increases.
- **Maturity of product categories:** Different product categories developed by one organization frequently are in different phases of the lifecycle. The challenge is that, depending on the maturity of a product category, the requirements on the common platform are quite different. For instance, for mature product categories cost and reliability are typically the most important whereas for product categories early in the maturity phase feature richness and time-to-market are the most important drivers. A common platform has to satisfy the requirements of all product categories, which easily leads to tensions between the platform organization and the product categories.
- **Unresponsiveness of platform:** Especially for product categories early in the maturation cycle, the slow release cycle of software platforms is particularly frustrating. Often, a new feature is required rapidly in a new product. However, the feature requires changes in some platform components. As the platform has a slow release cycle, the platform is typically unable to respond to the request of the product team. The product team is willing to implement this functionality itself, but the platform team is often not allowing this because of the potential consequences for the quality of the product team.

3. FIVE DIMENSIONS OF PRODUCT FAMILIES

In this section, we discuss the five dimensions that are of predominant importance in the context of broadening the scope of software product families, i.e. business strategy, architecture, components, product creation and evolution.

3.1 Business Strategy

Based on the author's experience in the domain of software product families, one can identify three predominant business strategies:

- **R&D minimization:** The first argument used by companies moving from product-specific to product family development of software is the reduction of R&D expenditure through the sharing of software artifacts between multiple products.
- **Time-to-market optimization:** Once the organization has successfully adopted a product family approach, the next argument often becomes the decreased time-to-market of new products as these products can share a significant amount of software.
- **Maximizing product family scope:** Once the organization is successfully bringing new products to market with agreeable R&D expenditure and time-to-market, there is often a drive to broaden the scope of the product family. This may be because the approach has proven its success in one product category and the organization is eager to build on this success by expanding it to other product categories. A second scenario is where the organization enjoys success in the market because of its adopted approach and is able to expand into new product categories. This article is concerned with the challenges and consequences of this third stage.

3.2 Architecture

In section 3.1, three, often consecutive, stages of business strategy are discussed. One can also identify four architectural approaches that are partially related to the stages discussed above.

- **Fixed structural architecture:** The first architectural approach, especially suitable for relatively narrow product families, is to specify a complete structural architecture as the basis for the product family. The architecture is the same for all products and variation is primarily captured through variation points in the components.
- **Micro-kernel architecture, optional elements:** An alternative, architecture centric approach is the combination of a micro-kernel, used for all products in the family, and significant set of optional elements that can be included, replaced or excluded depending on the product being derived.
- **Architectural principles guaranteeing compositionality:** The third approach does away with the structural architecture all together and focuses on the architectural principles that components have to satisfy in order to guarantee composability. This approach allows for the richest set of alternative configurations to be derived from the shared product family artifacts.
- **Accidental architecture:** Finally, in an excessive component-oriented approach, the architecture is the result of the opportunistic composition of independently developed components that share no or few architectural principles.

3.3 Components

Although architecture is very important and helps achieve business and operational qualities of software systems, it is of course the components that contain the actual implementations of functionality, features and requirements. Not surprising, however, the relation between the architecture and the components is more intimate than the terms may indicate. Again, we present three approaches to developing, managing and evolving components.

- **Internal integration-oriented components:** The first category is defined by the class of components that have been implemented specifically for a specific architecture that is specified in all or most of its aspects. The components contain variation points to satisfy the differences between different products in the family, but these do not spread significantly beyond the interfaces of the components. Finally, the components are implemented such that they depend on the implementation of other components rather than on explicit and specified interfaces.
- **Internal compositional components:** An alternative approach to implementing components is to develop components against explicitly defined provided, required and configuration interfaces and based on well-defined architectural principles. This approach allows for components that represent relatively independent domains of functionality and that can be freely composed with other components, as long as interfaces are adhered to and principles not violated.

- **External components:** In the era of open-source software, organizations often are able to satisfy a significant part of the requirements of a product through the selection of appropriate components. The resulting collection can often be complemented by commercially available components. This approach specifies an extreme approach, but in practice most product families define some form of common infrastructure consisting of external components. Thus, the latter two approaches are often combined.

3.4 Product creation

With all the focus on strategy, architecture and component, one would almost forget that the predominant reason for all this work is to cost-effectively and rapidly create a broad set of products. Again, here we can identify alternative approaches.

- **Product-specific code based on pre-integrated platform:** The integration-oriented model presented in section 2 typically assumes a pre-integrated platform that contains the generic functionality required by all or most products in the family. A product is created by using the pre-integrated platform as a basis and adding the product-specific code on top of the platform. Although not necessarily so, often the company is also organized is along this boundary. The approach works very well for narrowly scoped product families, but less well when the scope of the product family is broadening.
- **Composing components in product specific configurations:** An alternative approach is to rely on a composable set of components that can be relatively freely combined by a product team to compose a significant part of the functionality required by the product. The composition of reusable components can be interlaced with product specific functionality at any interconnect between two components as well as, architecturally, on top of the reusable components. This approach is based on the assumption that through enforcing architectural principles on the components relatively free compositionality can be achieved while maintaining system reliability.
- **Opportunistic integration and glueing of external components:** Finally, the third approach that can be pursued is the opportunistic integration and glueing of external components to create a product. Especially companies that, at some point, discover the open source software community and the tens of thousands of ongoing projects experiment with constructing systems through composing open source software components.

3.5 Evolution

The topic that, in my experience, often is the most challenging to manage well is the evolution of shared as well as product-specific software artifacts. Although the creation of an individual product based on reusable components brings benefit to the company, in practice the true benefits and cost appear when the whole machinery consisting of continuous product creation and the evolution and expansion of shared software artifacts is operating in normal mode. This dimension is primarily an organizational one as the evolution of features and requirements anyhow needs to take place from a technical perspective. Below, we discuss different alternatives that we have seen organizations use in practice:

- **Platform organization:** The first model, typically used in an integration-oriented approach is the strong preference towards incorporates new features and requirements into the pre-integrated platform. The reasoning behind this model is that new features, in due time, need to be provided in all products anyway and consequently, the most cost effective approach is to perform this directly. This instead of an alternative approach where product-specific functionality evolves and generalizes over time and is incorporated into the platform when the use of the functionality has spread sufficiently broadly.
- **New or extended components:** Especially in the case of a more composition-oriented approach, incorporation of new features/requirements occurs through the creation of new components or the extension of already existing components, frequently adding new variation points. Even new components, since these adhere to the architectural principles, can be composed with older components in cases where earlier a different component was used. The new component or the extension of an existing component can, depending on the organization,

be developed either by a component team or by the product team that requires the functionality the first.

- **Open-source community.** The third model, achieving increasing respect, is the proactive embedding of commercial R&D teams in the open-source software community. The important position in this case is viewing the R&D team to be part of the community, not outside it and collaborating with the community. When successfully implemented in a suitable domain, such a collaborative approach can lead to highly responsive and productive R&D work. Finally, it is important to mention that open-source software communities do not consist primarily of individuals that program for fun, but rather that several organizations may decide together that they care to share cost and effort required for the evolution of their collective products or systems.

4. BEYOND THE INTEGRATION-ORIENTED APPROACH

This paper is concerned with analyzing the challenges of broadening the scope of a software product family and by discussing alternative approaches to address these challenges. This section discusses two approaches. The first approach is the hierarchical product family. In this case, the reusable software artifacts are organized in a, typically, two-layered hierarchy. The first layer contains the generic code used by all products and the second layer contains the software artifacts that are used by all products in a specific category. The second approach is the composition-oriented method where the architecture is primarily principle-oriented and the components can be freely composed because these satisfy the principles. In the sections below, each approach is discussed in more detail.

4.1 Hierarchical Software Product Family

One scenario in the evolution of a software product family is that the product family naturally develops into a limited number of clusters of products. These clusters can subsequently be used as product categories and assuming that the size of each cluster and the amount of revenue generated are sufficient, business units can be made responsible for each product category. In this case, the logical approach is to organize the set of reusable components as well as the architecture of the product family in a hierarchical fashion. Thus, the functionality and features that are shared by all products in all product categories is developed as a platform by a shared R&D team. This platform is used as a basis by each business unit to build a software product family, again consisting of an extended architecture and a set of reusable components. Using the reusable product family artifact, the business unit can rapidly and easily develop new products and evolve new ones.

The hierarchical software product family occupies one point in the five dimensional space described in the previous section:

- **Business strategy - maximizing product family scope:** Although R&D cost and time-to-market are obviously relevant factors for any technology driven organization, the most important rationale for the hierarchical product family is that it facilitates the creation of a much broader set of products.
- **Architecture - micro-kernel architecture, optional elements:** The key challenge in the case of a hierarchical family is to architect the whole system such that that the business units can extend the platform architecture with the elements needed by their product category.
- **Components - internal integration-oriented components:** Although the hierarchical approach addresses several of the concerns discussed in section 2, it fundamentally takes the same integration-oriented approach as the original approach. The main difference is that the integration takes place in two stages, i.e. once for the basic platform and once for the reusable components for each product category.
- **Product creation - product-specific code based on pre-integrated platform:** As all reusable software artifacts are pre-integrated, product creation is primarily focused on the adding of product specific code on top of the reusable artifacts. This approach is excellent as long as the

product specific requirements do not affect the shared components, which is often the case in embedded systems.

- **Evolution - platform organization:** Although business units add product specific code on top of the shared components, it is the responsibility of the platform organization to bring new functionality and features to the business units. This evolution typically follows the vertical path, i.e. over time product-specific code matures and becomes part of the business unit specific shared artifacts. Subsequently, the functionality becomes part of the base platform and only then it becomes available to other business units. This means that there is no effective mechanism to share specific functionality between two business units.

4.2 Composition-oriented method

The second approach that we discuss in this paper is the composition-oriented method. The composition-oriented approach can be viewed as being positioned close to the one end of a continuum whereas the integration-oriented approach is close to the other end. The predominant factor defining the continuum is the balance between the organization developing the reusable software and the teams creating products using, among others, the reusable software. Depending on the organizational approach chosen, component teams may not even exist, as in the case of HP Owen [Toft et al. 00], and even if these exist, they are not necessarily part of a software reuse R&D team. The key challenge in this model is the architecture of the product family. As the architecture is based on principles, rather than a structural architecture, maintaining a consistent and productive architecture-centric environment is more difficult than in other approaches.

The composition-oriented approach can be pinpointed in one location in the five dimensional space presented earlier:

- **Business strategy - maximizing product family scope:** Although R&D cost and time-to-market are obviously relevant factors for any technology driven organization, the most important rationale for the composition-oriented approach is that it facilitates the creation of a much broader set of products.
- **Architecture - architectural principles guaranteeing compositionality.** As discussed earlier, the key difference between this approach and other approaches is that the architecture is not described in terms of components and connectors, but rather in terms of the architectural principles, design rules and design constraints.
- **Components - internal compositional components:** Obviously, the composition-oriented approach is very much driven by components. However, these components are not developed ad-hoc, but are constrained in their implementation by the architecture. As each component satisfies the architectural principles compositionality of these components is guaranteed.
- **Product creation - composing components in product specific configurations:** The explicit goal of this approach is to facilitate the derivation of a broad range of products that may need to compose components in an equally wide range of configurations. Product creation is, consequently, the selection of the most suitable components, the configuration of these components according to the product requirements, the development of glue code in places where the interaction between components needs to be adjusted for the product specific requirements and the development of product-specific code on top of the reusable components.
- **Evolution - new or extended components:** Product teams as well as component teams can be responsible for the evolution of the code base. Product teams typically extend existing components with functionality that they need for their product but is judged to be useful for future products as well. Product teams may also create new components for the same purpose. Component teams, if used, are more concerned with adding features that are required by multiple products. A typical example is the implementation of a new version of a communication protocol.

4.3 Analysis and Comparison

When analyzing the three approaches discussed in this paper, it is important to note that we have purposely excluded an open-source software based approach. The reason for this is that open-source software components can be included in each of the discussed approaches and as such are orthogonal to the approach chosen. However, the use of open-source software (OSS) has an impact on the software development organization as the evolution of OSS components can not be controlled to the same extent as internal components and most OSS licenses demand that the organization offers its addition to the OSS community. Nevertheless, despite these challenges, the use of OSS allows for very rapid creation of extensive software systems and can be a major factor in the reduction of R&D effort.

In the table below, we summarize the three approaches discussed in the paper. Each approach has a specific area of applicability. As discussed earlier in the paper for the case of the integration-oriented approach, applying an approach outside its area of applicability typically leads to several problems. As it for most organizations is difficult to replace an approach to software reuse that has been very successful in the past but has lost its applicability, this paper aims to discuss alternative approaches so organizations can, in a more explicit and objective manner, select the best approach. The table below summarizes the three approaches discussed in this paper.

Factors	Integration-oriented	Hierarchical	Composition-oriented
When applicable?	Well scoped family of highly related products	Broad family with a number of focused product categories	Broad family of products with significant unique requirements and features
Strategy	R&D cost minimization and/or time-to-market	maximizing product family scope	maximizing product family scope
Architecture	Fixed structural architecture	Micro-kernel architecture, optional elements	Architectural principles guaranteeing compositionality
Components	Internal integration-oriented components	Internal integration-oriented components	Internal compositional components
Product creation	Product-specific code based on pre-integrated platform	Product-specific code based on pre-integrated platform	Composing components in product specific configurations
Evolution	Platform organization	Platform organization	new or extended components
When used outside area of applicability	Unresponsiveness of platform leading to long lead times	Complicated alignment between hierarchical platform organizations leading to long lead times	High R&D cost due to significant overlap of development and integration efforts

5. RELATED WORK

This paper discusses two approaches that can be considered as alternatives to the integration-oriented approach that traditionally has been used frequently in the embedded systems industry. These alternative approaches are important because successful product families may broaden their scope considerably due to their success and this requires a conscious adjustment of the overall approach to software reuse. Although we believe that discussing these approaches in this context is a contribution of this paper, the approaches themselves have been discussed by other authors.

The paper by [Toft et al. 00] discusses the approach taken by Hewlett Packard's printer division where extensive sharing of software for the firmware of their products was achieved without the creation of a central domain engineering unit. A second author that has been promoting the notion of product populations and the consequences for the way software development is performed is Rob van

Ommering [Ommering 02]. These concepts were further explored in a later publication jointly with the author of this paper [Ommering & Bosch 02].

6. CONCLUSIONS

Software product families have found broad adoption in the embedded systems industry, as well as in other domains. Due to their success, product families at several companies experience a significant broadening of the scope of the family. This may be due to the fact that the cost of product creation has decreased significantly or because of management decisions that bring previously unrelated products under the umbrella of the product family. However, a broadly scoped product family requires a different approach than the traditional integration-oriented approach proliferated in many embedded systems companies. If the organization fails to adjust its approach, one can identify several problems that may result from this, including unresponsiveness of the platform, difficulties related to dealing with cross-cutting features and architectural challenges due to the need for over-engineering.

To address these concerns, we presented the five main aspects of product families that are relevant in this context, i.e. business strategy, architecture, components, product creation and evolution. Based on these dimensions, we have presented two alternative approaches. The first is the notion of hierarchical product families where the shared software artifacts are organized in, typically, two layers. The first layer captures the functionality that is common to all products in the product family whereas the artifacts at the second layer are specific to a product category. Secondly, we presented the composition-oriented approach that is different in that it is based on a set of architectural principles, design rules and design constraints rather than an architectural structure consisting of components and connectors. The components in this approach are relatively freely composable as these satisfy the same set of architectural principles. Finally, we analysed and summarized the three approaches discussed in the paper. The contribution of this paper is that it analyses the problems of and presents alternative approaches that are better suited for broad-scoped product families.

In future work, we intend to study the identified problems in more detail as well as evaluate the proposed alternative approaches more carefully as could be achieved in this paper. In addition, potentially further alternatives can be developed, especially approaches that cross organizational boundaries and/or involve the open-source software community.

REFERENCES

- [Bosch 00] J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach*, Pearson Education (Addison-Wesley & ACM Press), ISBN 0-201-67494-7, May 2000.
- [Ommering & Bosch 02] R. van Ommering, J. Bosch, *Widening the Scope of Software Product Lines - From Variation to Composition*, *Proceedings of the Second Software Product Line Conference (SPLC2)*, pp. 328-347, August 2002.
- [Ommering 02] R. van Ommering, *Building product populations with software components*, *Proceedings of the 24th International Conference on Software Engineering*, pp. 255 – 265, 2002.
- [O'Reilly 05] <http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html>
- [Toft et al. 00] Peter Toft, Derek Coleman, and Joni Ohta, *HP Product Generation Consulting, A Cooperative Model for Cross-Divisional Product Development for a Software Product Line*, *Proceedings of the First Software Product Lines Conference (SPLC1)*, Kluwer Academic Publishers, August 2000 (Ed. Patrick Donohoe), pages 111-132.