

# Improving product line development with subject-oriented programming

A position paper for ICSE2000 Workshop on  
*Multi-Dimensional Separation of Concerns in Software Engineering*

**Juha Savolainen**

Helsinki University of Technology  
P.O. Box 9700, 02015 HUT, Finland  
{Juha.Savolainen}@hut.fi

## Abstract

*It has been demonstrated the product lines have introduced large improvements to quality, time to market and overall productivity. However, creating a successful product line is a highly complex and difficult task. There are still many technological barriers to overcome in effective product line development. The current industrial practice employs patterns, idioms and components to handle complexity, but shortcomings in current object-oriented languages limit the effectiveness of product line development. Subject-oriented programming and more recently multi-dimensional separation of concerns promise improved support for product line development. Ideally, a product line can be composed of slices of an overall system that provide low coupling among components, good separation of unrelated concerns and improved understandability of the system structure. In this paper we describe our experiences on applying subject-oriented programming to product line development.*

## 1. Introduction

A company that produces a range of similar products has an opportunity to reduce the development, maintenance, support and marketing costs of each product by sharing some of the effort and parts between different products. In order to manage such sharing, related products are organized into families or product lines.

Transition to product line development process must be carefully planned. A company has to design its products so that the structure of the products allows sharing and so that development organization can utilize this commonality. Many successful companies have based their product development on the abilities of effective small teams and developed their products independently. Their success in the market place has created a situation where they maintain and further develop a large number of products that satisfy closely related requirements but do not share implementation at

all. To benefit from the similarities they have to redesign their product structure and base their product offerings on a common platform.

In this position paper we argue that using subject-oriented programming in a product family development, we can improve the quality and understandability of the underlying framework. This will maximize reuse potential by supporting extensions to the existing framework and therefore increase maintainability of the product family. In the same time we also discuss many obstacles that currently hinder using subject-oriented programming (SOP) [1] or hyperslices [2] for product line development. In section two we discuss how subjects and hyperslices separate concerns. Section tree describes major benefits that can be accomplished by utilizing SOP in the context of product lines. Several problems in current practices, tools and methods are identified in section four.

## 2. Separation of concerns

Separating different concerns in is an approach to divide the inherent complexity of the software into more manageable units. In an ideal world, these concerns could be investigated separately and then be integrated together to create a whole solution. However, these concerns are usually too closely related to be considered separately.

The separation mechanism in SOP or in HyperJ cannot provide total separation of concerns since a hyperslice or a subject may depend on some definitions that are made elsewhere. More specifically, composition rules [3] create the dependency between different concerns in the software system.

Hypeslices are considered to allow this separation by enforcing declaratively completeness. [4] Unfortunately, declaratively completeness does not separate concerns into truly isolated units, it just provides a method to define what kind of changes can be allowed to the referenced unit. In fact, there are a lot of implicit assumptions and definitions that are not included into

interface definitions. For example, these assumptions may relate to the performance of the component, on how messages are passed in the component, which kinds of optimizations are utilized or what kind of data is processed. None of these issues can be eliminated by the declaratively completeness in hyperslices. Notice that similar issues are valid to other software systems created in any commonly used paradigm. [5]

We need to make a less strict definition of separation of concerns. Generally, we cannot achieve separation of concerns into totally independent units. However, we can separate these concerns into units by providing a single location where a particular concern is located. This greatly enhances maintainability of the software intensive system and therefore assists evolution. Nevertheless, the possible changes may affect also the functionality of other hyperslices. This means that hyperslices provide only partial separation of concerns.

*Position 1. Separation of concerns, in the context of hyperslices and subjects, is an ability to structure a system in a way that the parts of the system, which contribute to the same concern, are localized into one unit.*

If we could totally separate every concern in a software system, we could isolate these concerns from every other aspect of the system. This would have a consequence that it would be possible to optimize a system in respect of any set of concerns or dimensions. For example, a developer might create an embedded system, which at the same time could achieve best possible performance, with lowest possible hardware resources, be highly maintainable while allowing great reuse possibilities. Practice has proved that creating this kind of system is very difficult. A company may optimize its products only for a few quality attributes but because of finite budget and time to develop the system, optimization to every possible parameter practically impossible. This issue is closely related to the *position 4*.

*Position 2. Hyperslices and subjects do not completely isolate different concerns or dimensions of concerns.*

### **3. Product line engineering with SOP**

Contemporary software engineering practices use encapsulation, frameworks, and product lines to manage complexity of software development. In many engineering domains natural tendency is to express entities and connections in a way that is most suitable to

the problem at hand. However, it seems to be hard, if not impossible, to create a single representation formalism that addresses all issues related to software engineering. Therefore, new view-based approaches to software engineering problems have emerged.

Many researchers have identified a need for views in requirement engineering [6,7], software architecture descriptions [8] and software specifications. These methods allow expressing views in different phases of software development, but mapping these concepts into programming language level has not been possible. Thus we have been forced to transfer our view-based design into object-oriented code that cannot express the separation of concerns that exists in the design specifications. Subjects and hyperslices allow expressing these issues in the language level, providing natural alignment from requirements and design to code.

If, during lifecycle of a system, it becomes clear that the system cannot deliver adequate performance, then modification to improve performance can require changes to the whole system. This kind of redesign can be very expensive and time consuming. We believe that using subject-oriented programming can help us to create more maintainable framework for product families.

In order to support traceability during the entire software lifecycle multiple levels of granularity are needed. Non-functional requirements (NFRs) generally express large, system-wide issues and goals that may not be visible in the language level. For example, it is useless to argue whether a pointer contributes to the performance or maintenance dimension. There is no one-to-one connection from the high level requirements to the single language statements.

*Position 3. Any low-level program structure contributes in multiple dimensions at the same time.*

Product lines constitute a great effort for a company. Multiple developers concentrate to create a reference architecture that will provide services and properties that are common to all current and future members of the product line.

Key issue in graceful evolution of product families is to maintain common reference architecture and prevent architectural decay. Expressing more stable domain characteristics in the core architecture increases maintainability of the product line by making the key assumptions more likely to remain constant during the evolution of the system.

The initial idea of composing the whole architecture from a set of subjects or hyperslices seems beneficial, since this would allow easier changes to the overall architecture, which by using other methods is more difficult to change. However, encapsulating NFRs in the

hyperslices is very complicated - if not impossible. Only some parts of the architecture seem to be suitable for encapsulation.

*Position 4. Architectural drivers of a system cannot be completely encapsulated in a hyperslice or a subject.*

Nevertheless, this does not render SOP or hyperslices useless. The possibility to partially localize these issues in a certain level of abstraction is an important feature. It allows considerable improvement in the understandability and maintenance of the system since using existing object-oriented methods these issues are distributed throughout the code.

One can encapsulate a reasonably large part of a concern into a set of composed hyperslices. This is needed because some issues are meaningfully visible only in a certain range of granularity or abstraction. Trying to localize these issues is useful and greatly can improve traceability and therefore evolution of the product line.

To our experience SOP can truly be used for expressing variance and improving evolution characteristics of product lines.

#### **4. Reuse**

There are several approaches for product line development with subjects. Applying hyperspaces adds new possibilities and methods to handle complexity that is inherent in product lines.

The main contribution of subject-oriented programming to the product line practice is the promise of easy evolution. Even though development methods try to estimate the possibility of change in requirements, unfortunately all possible changes cannot be predicted. Changes in the reference architecture of a product line are difficult because any modification potentially affects all instantiations of the product line. Applying subjects for these modifications makes it possible to experiment the consequences of the change. The change can be initially implemented as a subject, and if it is approved it can be included in the common code base.

Even if it is impossible to predict all future changes, a careful design starting from a domain model can dramatically reduce the number of modifications needed during the lifetime of the product line architecture. This means that subject-oriented programming is no substitute for good object-oriented design of product families, but it assists design by allowing more choices for variance encapsulation.

Object-oriented programming has been successful because data that is encapsulated in classes is more stable, than the methods that access the data. Similarly in

product lines a common reason for evolution is an introduction of new features. Product lines are created to satisfy demands for features in various market segments. If the environment where the products operate in is changing rapidly then the needs of the customers are likely to change, which may force a company to add more features in its products. Hyperslices and subjects allow encapsulation of a feature into its own unit removing it from the common framework. We believe that this kind of encapsulation is in certain domains highly important to guarantee support for maintenance of the system.

Implementing variance in features of the product family by subjects provides much needed flexibility to create products with different feature sets. Applying subjects allows that only those product line members that need a particular feature are composed with the corresponding subject. Then this particular feature is not present in the other products. Using conventional approaches this could be done by using parameterization, switches, or plug-in components. However with these methods similar degrees of separation, ease of use and flexibility that can be achieved with SOP cannot be easily reached.

Subjects or slices can also be used to allow different kinds of variations among product line members. Some product lines operate in a domain, which consists of multiple market segments with very different requirements. It is very difficult to satisfy requirements of a high-end system with strict performance requirements and on the other hand maintain compatibility with low-end systems, which run on minimum hardware requirements. Normal object-oriented practices do not help in designing these kinds of product families. In the case of subject-oriented programming it is possible to completely separate a feature with large hardware resource needs from the other system and therefore allowing creating low-end systems from the same product line.

Many product line development methods distinguish two different units – variable and commonality assumptions. However, in some application domains there are sub-groups within a product family that are very similar together and differ only with one or two respects. This fact allows increased reuse possibilities outside the common framework. Utilizing subjects or hyperslices in product lines allows easy composition and therefore makes it easy to benefit from these minor reuse possibilities.

Reuse in the product line initiative relies on the fact that the reference architecture is used in every product. But in the case of SOP it is impossible to enforce any architecture since there is no access control mechanism present in the language or in the tools that supports this approach. We believe that this is mainly tool

development issue that can be solved. But as it is implemented currently, one cannot prevent breaking the encapsulation.

One of the most promising applications of SOP in product line context is to use it for optimizations. By using subjects or hyperslices a developer can have some default functionality in the design of the current framework. A typical optimization scenario may require changes in multiple components or even a special purpose domain specific language to express complex parameterizations needed for different scenarios. However, implementing these optimizations with subjects reduces complexity and allows considering each optimization in isolation.

On the other hand, a large number of subjects/hyperslices distracts the overall picture of the system and therefore reduces the understandability of the system architecture. This is a major drawback. There is a great need for a tool that not only provides separation of the system into different slices but also allows composing these parts into a single architecture. Otherwise the total design of the system is vague and not easily understandable. There is a fine balance between separating crosscutting concerns into slices to improve understandability and increasing complexity by allowing too many hyperslices that interact in very complex ways.

There are many boundaries to reuse. Therefore a developer should feel that reusing existing assets is easy it will provide clear benefits. But also reuse practices should be enforced in a way that developers must use common components whenever it is practically possible. Unfortunately, hyperslices do not support access control mechanisms that are needed for any real world reuse environment. Since one may always override functionality that is defined in another hyperslice it is not possible to force anyone to reuse common code or design. This may lead to decay of system structure and increased complexity.

*Position 5. There is a great need for mechanisms that allows restricting access and controlling that e.g. common core architecture remains intact.*

## 6. Conclusions

Building a product line as a product family promises a high amount of reuse. To fulfill this promise product family development requires careful management. Most of the current methods require a developer to predict how requirements of the product family can vary. But there can always be situations that cannot be foreseen.

Subject-oriented programming offers new possibilities for the product family developer to deal with that kind of situation. We can more easily extend functionality of current frameworks and also modify them to satisfy requirement changes.

However, there are many problems that hinder the usefulness of the hyperslices and SOP. Some of these issues are identified and a few improvements have been proposed. We believe that subjects and hyperslices provide new possibilities for a product line developer.

In this paper, we have presented several claims to be discussed during the workshop. This paper represents the first ideas that have emerged from early experiments using SOP for product line development. Several issues have been presented that would improve the support for product line development even further.

## 7. References

- [1] W. Harrison, H. Ossher, Subject-Oriented Programming (A Critique of Pure Objects). Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'93), ACM Press, 1993, pp.411-428.
- [2] P. Tarr, H. Ossher, W. Harrison, S. Sutton, N Degrees of Separation: Multi-Dimensional Separation of Concerns, Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'99), ACM Press, 1999, pp.235-250.
- [3] H. Ossher, M. Kaplan, W. Harrison, A. Katz, V. Kruskal, Subject-Oriented Composition Rules. Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'95), ACM Press, 1995, pp.235-250.
- [4] P. Tarr, H. Ossher, Hyper/J User and Installation Manual, IBM, 2000.
- [5] D. Garlan, R. Allen, J. Ockerbloom, Architectural Mismatch: Why Reuse Is So Hard, IEEE Software, November 1995, IEEE, 1995, pp.17-26.
- [6] N. Nuseibeh, J. Kramer, A. Finkelstein, A Framework for Expressing the Relationships Between Multiple Views in Requirement Specification, IEEE Transactions on Software Engineering, October 1994, IEEE, 1994, pp.760-773.

[7] J. Grundy, Aspect-oriented Requirements Engineering for Component-based Software Systems, Proceedings of the Conference on Requirements Engineering (RE'99), IEEE, 1999.

[8] P. Kruchten,, The 4+1 View Model of Architecture, IEEE Software, November 1995, IEEE, 1995, pp.42-50.