Exploring Effects of Feature Mismatch to Evolution of Product Lines with Components and Aspects

Aleksandra Tešanović
Philips Research
The Netherlands
aleksandra.tesanovic@philips.com

Merijn de Jonge
Philips Research
The Netherlands
merijn.de.jonge@philips.com

Abstract

We explore the observation that the evolution of feature space and the evolution of components lead to a fundamental mismatch between features and their implementation in components. At one moment in time, a software system is structured in components in such a way that a given feature set can be mapped straight-forward to these components. Over time features and components evolve, leading to a situation where the mapping is less straightforward. A common indication is the need to deal with crosscutting features. In the end, the component structure that should improve software development hampers development because the implementation of individual features gets too much scattered around different components. We claim that using aspects for dealing with feature mismatch is beneficial in a short-term but can introduce additional problems in a long run, especially if aspects are used to mask architectural problems.

1 Introduction

With the emergence of product line architectures and technologies such as aspect-oriented and component-based software development, classic categories of evolution, including *corrective* (to fix errors), *adaptive* (to adapt to changing environment, such as changed requirements), *perfective* (to enhance the software), and *preventive* (to improve sustainability) [8], have been augmented. Namely, the evolution has also been examined over time and classified as *continuous* and *discontinuous* [3]. Continuous evolution corresponds to changes in the software within an existing software architecture, while discontinuous evolution implies extensive changes to the software to accommodate new requirements, which normally result in an emergence of a new architecture. In this context each architecture has a tolerance level, which it can sustain with continuous evolution [3]. Above that level, the price of adding new changes into the existing architecture is more expensive than creating a new architecture. The evolution is therefore disrupted and a new architecture for a product line is defined. Longevity of the architecture is in proportion with the tolerance level, and this tolerance level depends directly on the cost associated with evolution, e.g., the cost of adding unplanned new features. The higher the tolerance level, the longer it is possible to reuse the architecture (and reuse is one of the drivers behind product line architectures [2]). In parallel to the time dimension of software evolution another dimension has been identified, where, depending on the effect of the evolutionary changes to the architecture the evolution is classified as either *crosscutting* or *non-crosscutting* [9]. Changes affecting more than one component of the software architecture are considered crosscutting, while the localized changes are considered non-crosscutting.

Our experience showed that (eventually) product line evolution consists of a series of continuous crosscutting changes followed by a disruptive change. Disruptiveness results in a new architecture that more easily facilitates incorporating emerging requirements of new products and consequently incurs less costs when adding new features. The costs of disruptive changes are balanced with the additional costs associated with defining a new architecture and an effort to implement that architecture. Crosscutting changes intrinsically do not cause disruption of continuous evolution. However, it is our observation that a number of accumulative crosscutting changes are the key factor for disruption of continuous architecture evolution. One of the reasons the accumulated crosscutting evolution has such profound effects on the overall evolution process is the phenomenon of feature mismatch, which we observed during Philips TV product line evolution.

In this paper we explore the feature mismatch phenomenon and its origins, and discuss under which conditions one should opt for aspect-oriented solutions in the context of product line evolution. We elaborate that aspects, if

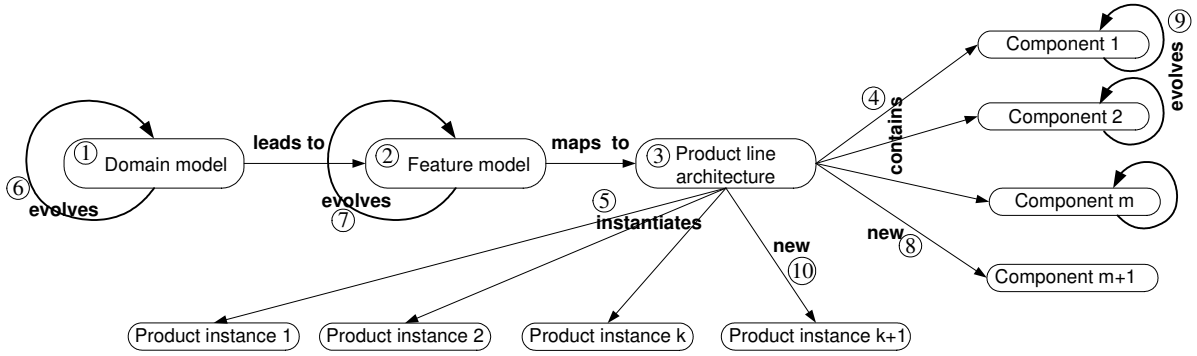

Figure 1. The evolution process of a product line.

not administered carefully into the product line architecture, could eventually result in increased costs of the architecture evolution.

Moreover, we introduce an additional classification of software evolution *linear* evolution. Linear evolution is an idealized, cost-optimal evolution pattern where the normalized cost associated with adding a new feature is constant over time, rather than non-linearly increasing. We elaborate on characteristics and benefits of linear evolution, its relation to cost and feature mismatch, as well as give hints on how the linear evolution could be achieved.

The paper is organized as follows. In section 2 we explain the feature mismatch phenomenon. The role of aspects in the context of evolution and feature mismatch is discussed in section 3. Section 4 concludes the paper with the discussion about linear evolution and steps to achieve it.

2 Feature Mismatch in Product Lines

Here we discuss the evolution of a product line architecture and explain how accumulated crosscutting evolution can result in a feature mismatch and increased costs.

2.1 Creating feature mismatch with components

Figure 1 depicts the process of product line architecture creation and evolution. The first step in product line architecture creation is domain modeling, where a domain is assessed and used as input to the product scoping and feature modeling. Then, based on the required features in the product line, the overall architecture is defined in the following step. Each product line instance is created by defining and composing a number of components (step 4 in the figure). Components are then used within the architecture for instantiation of a number of different product instances. This is indicated by step 5, which shows that from a product line architecture, using components $1, \ldots, m$, one can create products $1, \ldots, k$.

When the domain or a product scope changes, e.g., new features need to be integrated into the product line architecture, a complex evolution process is triggered (depicted in steps 6 – 10). The evolution starts with a revision of the domain and is followed by the the evolution of the corresponding feature model, as depicted in step 7.

Evolution for the feature model normally entails adding new features or modifying existing ones. In practice, implementing these features requires adding new components (see step 8 where a new component $m + 1$ is added), and possibly modifying existing components (step 9). Finally a new product instance can be obtained from the new set of components.

From the evolution process and its impact on components and architecture, we observe that the evolution of feature space and the evolution of components lead to a fundamental mismatch between features and their implementation in components (see Figure 2). At the time of architecture creation, a software system is structured in components (C) in such a way that a given feature set (F) can be mapped straightforwardly to these components. Over time features and components evolve (into feature set F' and component set C', respectively) leading to a situation where the mapping is less straightforward. Feature mismatch refers to the situation that during product line architecture evolution the feature set evolves independently of the component set, resulting in the link between evolved features and component to become obscured. A common indication is the need to deal with crosscutting features. That is, adding new features has impact on an increasing number of components, rather than on a small set of related components.

2.2 Measuring feature mismatch

The effect of the mismatch of mapping features to components can be observed when studying interface evolution. Ideally, interfaces are invariants over evolution in order to

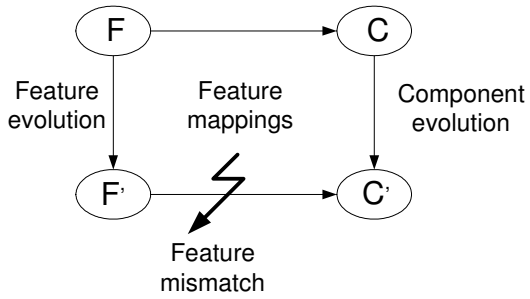

Figure 2. Feature mismatch.

preserve benefits of using components. They only change under strict conditions (e.g., following interface evolution rules [13, 14]). Such rules enforce that under normal circumstances, adding a feature should not significantly affect interfaces of existing components.

If adding a new feature has crosscutting effects on many components, this can be observed by analyzing interface evolution, since the change to the component will (most likely) be reflected in an interface change. If adding a feature leads to a chain of component changes, this would signal coupling between components. Since components are supposed to be units of independent deployment [11], (strong) coupling between components often forms an indicator that the decomposition of a software system in components is suboptimal. Hence, interface changes may provide information about how well a system's structure in components matches its features. Monitoring interface evolution over time gives insight in how feature mismatch is emerging.

We explored the effect of feature mismatch on the concrete example of the Philips TV product line, which consist of four main reusable subsystems [13]: i) the platform subsystem provides a stable API that abstracts TV-specific hardware; ii) the infrastructure subsystem provides an abstraction of the operating system functionality, and also handles time-critical operations; iii) the services subsystem is a middleware layer between the platform and the applications; and iv) the applications subsystem includes the user interface for various TV applications, e.g., program installation, audio control, and video control, as well as the rules for avoiding undesirable interaction between these applications.

What we observed is that on average 11% of interfaces suffered from changes in various parameters. The most affected system is the service subsystem with as much as 44% interface changes. The application subsystem also suffered with approximately 33% of changed interfaces. The infrastructure subsystem has 16% interfaces changed and the platform is stable with approximately less than 3% of changed interfaces. The reason that the service subsystem is the most

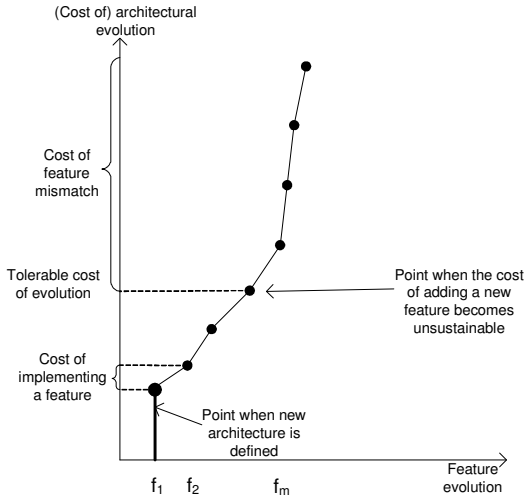

Figure 3. Theoretical cost development of implementing new features in the architecture.

effected one is that most of the core TV functionality is located in this subsystem, and any change in the application, infrastructure, and platform subsystem is also most likely propagated to the service subsystem.

To confirm that this amount of interface changes was indeed a symptom of feature mismatch, we chose an arbitrary feature set (related to multi-window control) and explored its evolution in the context of component and interface changes. We observed that it is straightforward to determine a component and its interfaces that were initially defined for (an initial set of) multi-window control features, i.e., the component name had multi-window in its name and/or description. Two interfaces of this component were modified during the evolution of the multi-window feature set. Following these changed interfaces, we further revealed that the change in the multi-window component, which we took as a starting point of the investigation, has affected interfaces in three out of four subsystems and consequently propagated to a majority of components within each of the affected subsystem. The multi-window feature that has introduced the change was difficult to determine in the modified interfaces. Due to so many changes in the interfaces of various seemingly unrelated components, only through a very throughout inspection and extensive domain knowledge it was possible to link the change in the interface and the component to the change in the multi-window feature set.

2.3 Cost of Feature Mismatch

In the end, the component structure that should improve software development (e.g., by shortening time-to-market,

decrease maintenance cost etc.) hampers development because the implementation of individual features gets too much scattered around different components. This has a direct and negative manifestation on the cost of adding a feature, as illustrated in Figure 3. When an architecture is defined (see the first point in Figure 3) the initial set of features is implemented and the architecture is optimized for that particular set of features. This implies that the architecture can sustain adding a new feature at another point in time with optimal costs (second point in Figure 3). The cost of implementing new features will increase over time. At some point the cost of new features becomes so high that it is no longer cost-effective adding them. At that time, the feature set may be frozen, making the product less competitive. Alternatively, a disruptive evolution step can be initiated to create a new architecture that is better tuned towards the feature set of that moment. Again, this architecture is fine-tuned for adding features that are foreseen in the near feature. These can be added with acceptable costs. As the time elapses and new features are added to the software, the cost of adding a feature increases again to a point where it is more cost-effective to engage a disruptive evolution process and create yet a new architecture.

The overall process of evolution is depicted as the series of lines between the points where new features are added by means of continuous evolution or by discontinuous evolution (when new architectures are defined). The effect of feature mismatch is that evolutionary changes of the software (crosscutting and continuous) result in an architecture that no longer can accommodate these changes at an acceptable cost, as indicated in Figure 4.

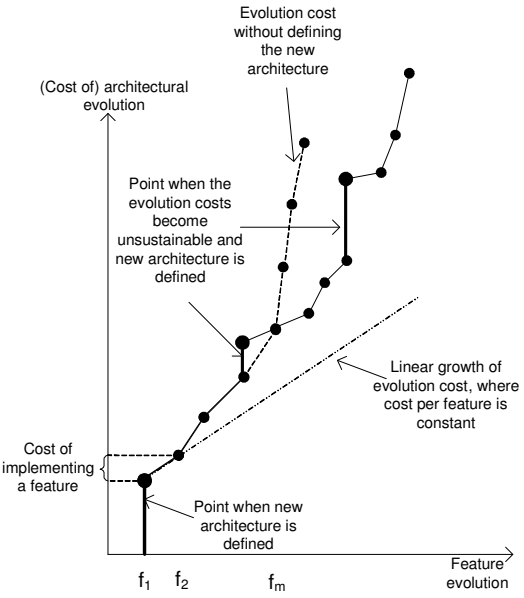

Figure 4. Theoretical cost development of evolution when new architectures are defined to ensure lowered costs.

3 The Role of Aspects to Feature Mismatch

Aspects have been introduced as a way to cope with crosscutting architectural and evolutionary issues in product line engineering [9]. They enable encapsulating crosscutting concerns of the system (such as features) into aspects that can be developed, maintained, and evolved independently of the (core) system; hence, separating concerns in the system and the architecture. As we already mentioned, and has earlier been reported [9, 12], many evolutionary changes are crosscutting. Most of the features, for example, that are used to evolve the Philips TV product line are crosscutting by nature, e.g., changing a platform results in changes in other part of the architecture.

Introducing aspects in an existing product line architecture is attractive because it does not require to restructure the system. This has a technical advantage since the structure in components does not need to be altered. As the component structure often reflects the structure of the organization (or the other way around), it also has an organizational advantage because the existing structure of development teams etc. can remain unchanged. It is therefore far more easy (at first) to start using aspects, then to optimize the structure of the system and organization to minimize the crosscutting effect of a feature.

However, the point when an organization contemplates introducing aspects in their product line is often the point when crosscutting changes have become so expensive that a new way of organizing software becomes an obvious necessity. At that particular point introducing aspects has immediate potential as it promises to rapidly decrease the cost of crosscutting evolution of the software (see Figure 5). Despite this advantage on the short term, we claim that the use of aspects for implementing features can quickly become a drawback. That is, with the increase in number of features the use of aspects can result in a very steep increase in costs (see Figure 5). The increase in cost can occur for at least the following three reasons:

- Aspects are used to treat the symptoms not the ”disease”. This is the case when an organization turns to aspects as a way to deal with evolutionary problems that hamper cost-effective development of new features. We claim that if introducing aspects is done without serious introspection of the internals of an architecture, then their introduction can mask and hide architectural problems, which then manifest themselves more strongly at a later point in time. Namely, if aspects are used to hide architectural problems, they will only delay the decision to make a new architec-

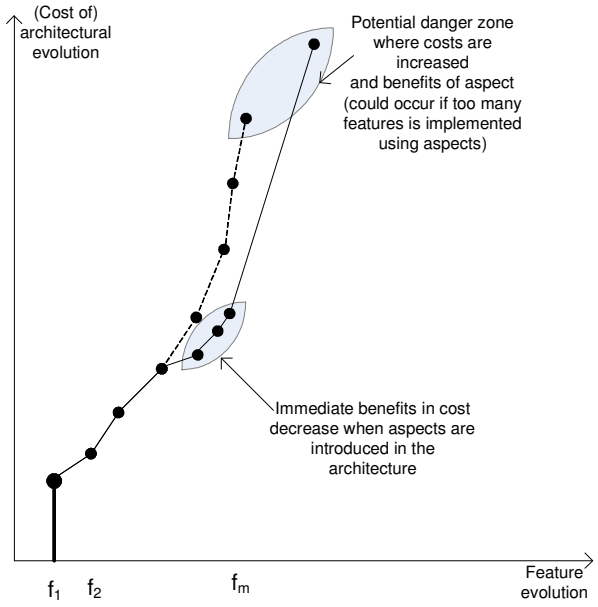

Figure 5. Theoretical cost development of evolution with aspects.

ture and when the decision is made will introduce more costs for creating a new architecture.

- Aspects are developed separately but integrated globally. The costs associated with evolution is also due to the increased effort in integrating the features with the overall system, and the testing activities that need to be done on the system level. The advantage of (component-based) product line is that, ideally, testing on the component level decreases the efforts associated with testing on the system level. With the introduction of aspects this advantage cannot anymore be exploited as, to the best of our knowledge, there is no industrially mature approach that enables modular checking of aspects and components.

- Organization of feature development as crosscutting aspects is expensive, because it is difficult to align the development with all components involved, and with the corresponding development teams. The fixed allocation of development teams, as it is currently the case in industrial product lines, is also in part done to maintain domain knowledge (as well as because component internals are so complex that it is difficult to reallocate teams). The larger the number of aspects, the more difficult their development becomes, as well as determining their relation with components in the system. To make things worse, crosscutting features may start crosscutting other crosscutting features, such that the corresponding aspects crosscut other aspects.

We argue that the need for aspects in a product line architecture can be an indicator for the quality of the architecture. A growing need for aspects over time can indicate that the structure of the product line architecture is no longer optimal, and may need revisiting. For example, observing an increase in changes to interfaces may signal that the product line requires more crosscutting changes. Rather than introducing aspects develop the crosscutting features, it might be the right time and, in the end more cost-effective, to consider refactoring the system.

The use of aspects in product lines to deal with crosscutting features is well being studied, see e.g., [7, 6, 10]. These papers describe the design of product lines with aspects and address typical problems with aspects, like increased coupling. Consequently, their solutions will help to improve the practice of aspect-oriented product line design, and help to reduce cost of developing crosscutting featues. However, the evolutionary pattern that may be the casue of increased crosscuttings is not addressed.

Our observations are not intended to discourage the use of aspects. On the contrary, we believe that aspects can be very powerful. Our aim is to signal that aspects, as components, should be used with care (which is in line with e.g., [10, 6]), and only after the real need for them is apparent, i.e., that they are not used for hiding architectural problems.

4 What's Next?

As discussed, the evolution of a product line architecture traditionally followed a crosscutting, continuous-disruptive pattern. Although disruption enables costs to be decreased in a short-term, the overall increase of costs is still far from being optimal in a long-term. From Figure 4 we can observe that the implementation of the first feature after the architecture is defined will be optimal, but also that this cost quickly increases with new features. To keep the cost of adding features low, the evolution cost should follow a linear pattern, as indicated by the optimal cost line in Figure 4. Anything above the optimal cost line indicates cost as a result of feature mismatch. Ideally, if evolution strictly follows the linear pattern, the normalized cost of each increment is kept constant.

From the structural point of view this implies that an architecture should be flexible, way beyond the (re) configurability that is currently possible with components and aspects. The structure of the system should have fluid properties to be able to be continuously reconsidered. This also implies that features can be assigned more dynamically to different development teams. This is in contrast with current practice, where component structures are so difficult

to change [5] and where the use of aspects to mask structural problems further increases the architecture's complexity. Hence, despite the promised benefits of components and aspects, they can also have a detrimental effect to an architecture's evolution.

Although we don't have concrete solutions yet, the following observations may lead in the right direction.

- Structuring a system in components and having aspects that crosscut this structure is positive for the implementation of the instances of a product line at a given moment in time. This is because reusing (i.e., sharing) code between instances of a product line is typically beneficial. The different instances of a product line can benefit more from sharing code, because the cost for implementing a new feature has to be made once and can then be shared across the different instances. However, sharing code over time (i.e., between different versions of a product) might not be so effective, because of the feature mismatch problem, as discussed in this paper.

- As an alternative to sharing code between product versions, sharing more abstract artifacts or domain knowledge (such as algorithms and design) could bring additional benefits, since we do not want to loose gained expertise between versions. Thus, rather then reusing the implementation of an algorithm which is tailored for version x, and hard to adapt for the requirements of version $x + 1$, it would be beneficial if we could reuse the algorithm and derive a specific implementation for version $x+1$. By reusing the algorithm, we do not loose expertise about the algorithm, only its implementation.

For reuse between product instances current technology can be used, such as components aspects etc. Reuse of knowledge over time relates to model-driven architectures, where different models describe different aspects of a system. Ideally, model-driven architecting can be used to automatically derive an implementation from a set of models. Recently approaches that explore the combination of aspect-orientation with model-driven architectures in product line domain have emerged, e.g., [1, 15, 4]. However, an open question still remains as to what is the proper abstraction level to make these models reusable without introducing feature (model) mismatch.

Another challenging question is whether it would be possible to (automatically) derive an optimal component structure from a selection of features describing (functional and non-functional properties of) a product and a collection of knowledge, e.g., the knowledge can contain existing designs, algorithms, and architectural implications. This would enable a fluid architecture that is able to sustain the pressure of evolutionary changes with minimal cost.

References

[1] AMPLE project: Aspect-oriented, Model-driven, Product Line Engineering. Project website: http://ample.holos.pt/, 2007.

[2] Hunting product-line architecture. Software Engineering Institute (SEI), 2007.

[3] M. Aoyama. Continuous and discontinuous software evolution: aspects of software evolution across multiple product lines. In *Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE'01)*, pages 87–90, New York, NY, USA, 2001. ACM Press.

[4] A. Carton, S. Clarke, A. Senart, and V. Cahill. Assessment of product line architecture and aspect-oriented software architecture methods. In *Proceedings of the First Workshop on Aspect-oriented Product Line Engineering (AOPLE-1))*, 2006.

[5] M. de Jonge. Build-level components. *IEEE Transactions on Software Engineering*, 31(7):588–600, July 2005.

[6] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, pages 51–60, Jan./Feb. 2006.

[7] U. Kulesza, V. Alves, A. F. Garcia, C. J. P. de Lucena, and P. Borba. Improving extensibility of object-oriented frameworks with aspect-oriented programming. In M. Morisio, editor, *ICSR*, volume 4039 of *Lecture Notes in Computer Science*, pages 231–245. Springer, 2006.

[8] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21(6):466–471, 1978.

[9] L. Neil, R. Awais, W. Zhang, and J. Stan. Supporting product line evolution with framed aspects. In *Proceedings of the Fifth International Workshop on Product Family Engineering (PFE-5)*, Siena, Italy, november 2003.

[10] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information hiding interfaces for aspect-oriented design. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 166–175, 2005.

[11] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1999.

[12] A. Tesanovic. Evolving embedded product lines: opportunities for aspects. In *ACP4IS '07: Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software*, page 10, New York, NY, USA, 2007. ACM Press.

[13] R. van Ommering. *Building Product Populations with Software Components*. PhD thesis, Rijksuniversiteit Groningen, The Netherlands, 2004.

[14] R. van Ommering. Software reuse in product populations. *IEEE Trans. Softw. Eng.*, 31(7):537–550, 2005.

[15] M. Voelter and I. Groher. Product line implementation using aspect-oriented and model-driven software development. In *Proceedings of the 11th International Software Product Line Conference (SPLC 2007)*. IEEE Press, 2007.