

Concerns of Variability in “bottom-up” Product-Lines

Andreas Speck, Matthias Clauss, and Bogdan Franczyk

Intershop Research Jena

D-07740 Jena, Germany

[a.speck | m.clauss | b.franczyk]@intershop.com

Abstract. Conventional product-line approaches tend to implement product-lines from scratch rather than support the complete reuse of existing systems. In contrast to these “top-down” approaches the “bottom-up” procedure proposed in this paper allows reuse. The modification and extension with variable elements is realised by aspects woven into existing systems. A set of different concerns provide the different potential specialisations of the variability. Moreover aspects may be used to implement the dependencies between different flexible elements.

1 Introduction

The intention of software reuse is rather old. Already at the NATO science conference on software engineering D. McIlroy stated [12]:

“Software engineers should take a look at the hardware industry and establish a software component subindustry. The produced software components should be tailored to specific needs but be reusable in many software systems.”

One way to reuse code which comes McIlroy’s vision very close is product-line software engineering. A common procedure is to start with intensive domain engineering in order to capture the requirements, features and variability of a product-line to be developed. The precondition of this conventional approach is that the system to be build has to be build from scratch. If a system is already existing as base for the product-line this systems has to be re-engineered which usually bears a similar effort as developing a new system.

2 “Bottom-up” Product-line

2.1 “Bottom-up” Product-line Development Process

In this paper we would like to introduce a concept in order to design the variability without a complete re-engineering of an existing system. Such an approach allows to consider large systems like e-marketplaces as a potential target for product-line engineering. We would like to call this concept the “bottom-up” approach of product-line engineering.

In principle the compliance with the existing “top-down” approaches is that in both cases extensive domain engineering is the base for the development or derivation of the product-line system. Domain engineering allows to define the *hot spots* of the

product-line in contrast to the *frozen spots*¹ which have to be reused as is. On base of the definition of the required variable elements we start to derive a product-line.

The “bottom-up” approach uses existing systems as base of a product-line in contrast to “top-down” product-line development. The product-line is realised by adding the *hot spots* modelled and designed by the domain engineering to the base system. Like in common approaches the product-line may then be transformed into many different systems by specialisation and customisation of the *hot spots*. In our approach the *hot spots* are considered as cross-cutting concerns.

2.2 UML Notation for Feature Modelling

The most common approach to model features and the variability of product-lines is applying the FODA model. Despite the wide spread knowledge of the FODA notation among researchers it is not very well-known in the world of software development. System Developers are mostly not familiar with FODA although they are experienced in other notations describing object-oriented models like UML. This led us to the conclusion that it might be much more useful to apply the well-known UML in order to model the features and variability [6].

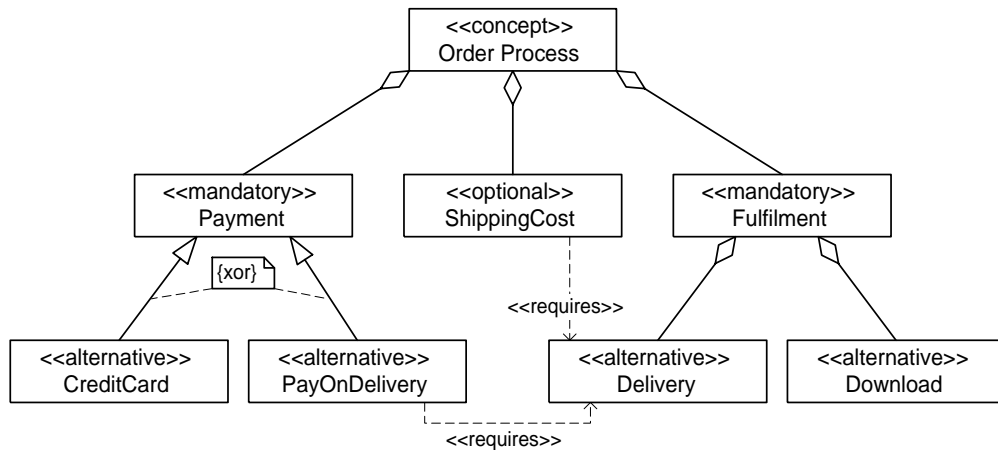


Fig. 1. UML Feature Model: *Order Process* of an eCommerce System

Figure 1 depicts the simplified feature model of a order process that is part of almost every eCommerce system. The overall concept *OrderProcess* is decomposed into the features *Payment*, *ShippingCost* and *Fulfilment*. *Payment* is specialised in several payment possibilities: by *CreditCard* or *PayOnDelivery*. As you can select only a single method per order these are modeled as exclusive alternatives. In opposition, the fulfilment strategies *Delivery* and *Download* can be co-existent, e.g. offering a software per *Download* and sending a copy on CD via the slower postage delivery. *Delivery* is also an precondition for *PayOnDelivery*, modeled with a ‘requires’ composition rule. Also *ShippingCost* only apply if you use *Delivery*. Therefore *ShippingCost*’s are optional since they don’t have to be paid of only *Download* is used.

In contrast to a FODA model in our “bottom-up” approach an UML feature model may be simply derived from models of the existing non-product-line system. Actually

¹ The terms *hot spot* and *frozen spot* have first been introduced in the domain of frameworks [14].

the UML feature model is an abstract model of the system as it may be used in the analysis phase in order to define the requirements (in fact the feature model represents the requirements for the product-line system). This abstract model has now to be extended by the various alternatives, dependencies and exclusions which define the potential realisations of the product-line system.

3 Variability of Systems

There is no formal pattern how the feature definition could be automatically mapped to the architecture. Like in object-oriented notations (e.g. UML) there is a certain semantical gap which makes it impossible to generate the derivation.

Since in our “bottom-up” approach the invariable elements (*frozen spots*) are already known only the *hot spots* have to be addressed. This could be done in a traditional way or with aspects. The latter seems to be the natural way to solve the problem.

3.1 Traditional Approaches

In [17] H.A. Schmid demonstrates how *hot spots* may be designed by applying design patterns which is actually an conventional approach which preserves most of the existing architecture. He determined that e.g. 19 of the 23 design patterns of the GoF design pattern catalogue may be applied to model variability. Most of these patterns are based on the *Strategy* pattern [8]: An abstract super-class defines an interface which has to be kept by the real sub-classes. The sub-classes override the inherited abstract methods and modify the architecture by this way. This approach assures that the real sub-classes fulfil the requirements given in the super-class where the sub-classes may be variably exchanged. When a sub-class doesn't fit to the scheme (interface) determined by the super-class the compiler will realise the problem (e.g. the signature of an overriding method doesn't fit to the abstract method or there is no method given to override an abstract method in the super-class which is called by the architecture).

The application of design patterns is very useful and eases the development of variable *hot spots*. However the disadvantages of this approach are that a modification requires a new class or object to be explicitly integrated by inheritance into the system. It would be much better if the classes to adapt the product-line existed already in the product-line and are simply switched on or off. Moreover there exists no possibility to define in advance negative (or positive) interactions between the features to modify the product-line. This problem could be solved by applying the concept of cross-cutting concerns.

3.2 Implementing Variability as Cross-cutting Concerns

Our intention is to modify the existing base system by adding new items (realising the variability). In general these new elements may be either:

- filters that switch on or off specific existing functionality
- or predefined functionality which may be arbitrarily added to the system.

Both cases may be realised by applying aspects.

Figure 2 depicts the implementation of the variability by aspects. The aspects have been woven in the existing base system and serve as filters and connection mechanisms between the classes. They may be woven to classes which are already existing in the base system as well as to classes which are added in order to build a product-line (like

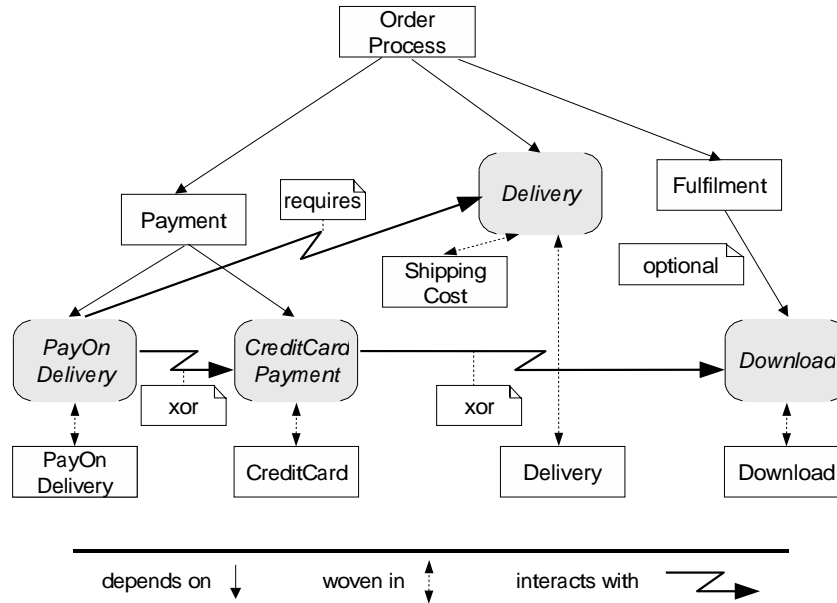


Fig. 2. Aspect Model of Order Process Variability

an additional way fulfilling customers order, e.g. *Download* if *Delivery* exists already in the base system).

The applicability of aspects is given since aspects may be used to implement design patterns. However the aspects may be simply woven into the existing system in order to implement a variability. The modification of the code like the traditional implementation of flexible elements propose (c.f. section 3.1) is not required. Moreover aspects allow to integrate several classes or subsystems depending on each other. In the *Order Process* example *Delivery* and *Shipping Cost* are both integrated with the same aspect in the system.

In contrast to simple design patterns interactions (*requires* and *xor*) between different *hot spots* can be modelled and implemented as well. These interactions are indicated by a jagged arrow. When a system is derived from a product-line the interaction have to be taken into account in order to prevent errors. Further aspects may be used to implement the interactions. These aspects assure that the required interaction is performed. A version system like proposed in [15] may support the design and validation of a larger number of interactions. In this case versions express the different possible customisations of the *hot spots* which are correct.

Since there exist different aspect concepts to realise aspects it is one goal of our approach to define aspect systems which support this “bottom-up” product-line approach. Currently we investigate a transformation system operating on the abstract syntax tree of a product line. Quite close alternatives may be the composition filter approach [5], adaptive component connectors [13] or AST manipulation approaches like [18].

4 Related Work

Complementary work may be found in all aspect-oriented and related approaches like ASPECTJ [10] subject-oriented programming [9], adaptive programming [11] or also transformation systems [3] (including the above mentioned adaptive plug & plays [13],

composition filters [1] or XML-based AST manipulations [18]). All these approaches may potentially be applied to realise the “bottom-up” product-line approach. They support the implementation of the “bottom-up” way of inserting variability into existing systems. Additionally generative programming [7] may provide means to generate variable elements into base systems which is yet another realisation for separated concerns.

In contrast to existing product-line engineering methods the “bottom-up” approach is oriented on the reuse and extension of existing systems. However there had been approaches to apply aspect-oriented principles to build product-lines. One approach is GenVoca [2] which allows to build a system family as layered architecture. Another approach may be found in [4]. This paper proposes the application of aspects in product-line software development in general.

Complementary to the “bottom-up” approach is the field of *feature interaction* which provides means to ensure that only correct flexible elements exist in a product-line and that only valid systems may be derived from this product-line. Several approaches have been discussed at the ECOOP 2001 Feature Interaction in Composed Systems workshop [16]. A potential solution for the modelling and verification of the static dependencies between features (and the aspects realising the features as well) may be the versioning approach introduced in [15].

5 Conclusion

The “bottom-up” product-line approach is invented in order to support the reuse of complete existing systems. These systems are transferred to product-lines by the weaving in of concerns which implement the variability (or a chosen customisation respectively). Therefore the high effort for re-engineering existing systems or even rebuild like in conventional approaches (“top-down” approaches) may be considerably reduced.

The aspects realising the variability serve as filters or connectors between classes and subsystems and the static part of the product-line (*frozen spot*). Additionally the interactions between the flexible elements may be realised by cross-cutting concerns.

References

1. M. Aksit. Composition and Separation of Concerns in the Object-Oriented Model. *ACM Computing Surveys*, 28(4), December 1996.
2. D. Batory and B.J. Geraci. Composition Validation and Subjectivity in GenVoca Generators. In *IEEE Transactions on Software Engineering*, pages 67 – 82, 1997.
3. I. Baxter. Design Maintenance Systems. *Communications of the ACM*, 35(4):73 – 89, April 1992.
4. J. Bayer. Separation of Concerns in Product Lines Engineering. In K. Mehner, M. Mezini, E. Pulvermüller, and A. Speck, editors, *Aspektorientierung - Workshop der GI-Fachgruppe 2.1.9 Objektorientierte Software-Entwicklung*. Technischer Bericht der Universitt Paderborn tr-ri-01-223, May 2001.
5. L. Bergmans and M. Aksit. Composing Crosscutting Concerns using Composition Filters. *Communications of the ACM, Special Issue on Aspect-Oriented Programming*, 44(10):51 – 57, October 2001.
6. M. Clauss. Generic Modeling using UML extensions for variability. In *Workshop on Workshop on Domain-specific Visual Languages, OOPSLA 2001*, pages 11–18, Tampa, Florida, June 2001.
7. K. Czarnecki and U.W. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, 2000.

8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Abstractions and Reuse of Object-Oriented Software*. Addison-Wesley, Reading, 1994.
9. Osshier H. and P. Tarr. Using Subject-Oriented Programming to overcome common Problems in Object-Oriented Software Development/Evolution. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 687 – 688, May 1999.
10. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *LNCS 1241, ECOOP*. Springer-Verlag, June 1997.
11. K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.
12. M. D. McIlroy. Mass-produced software components. In P. Maur and B. Randell, editors, *Engineering Concepts and Techniques, Proceedings of 1968 North Atlantic Treaty Organisation (NATO) Conference on Software Engineering Garmisch-Partenkirchen*, pages 138 – 150. NATO Science Committee, Jan 1976.
13. M. Mezini and K.J. Lieberherr. Adaptive Plug-and-Play Components for Evolutionary Software Development. In *ACM SIGPLAN notices*, volume 33, October 1998.
14. W. Pree. *Komponenten basierte Softwareentwicklung mit Frameworks*. dpunkt, Heidelberg, 1997.
15. E. Pulvermüller, A. Speck, and J.O. Coplien. A Version Model for Aspect Dependency Management. In *Proceedings of International Symposium of Generative and Component-based Software Engineering (GCSE 2001)*, LNCS 1241, pages 70 – 79. Springer, 2001.
16. E. Pulvermüller, A. Speck, M. D’Hondt, W.D. De Meuter, and J.O. Coplien. Workshop on Feature Interaction in Composed Systems, ECOOP 2001. Budapest, Hungary, June 2001.
17. H.A. Schmid. Systematic Framework Design by Generalization. *Communications of the ACM*, 40(10):48 – 51, Oktober 1997.
18. S. Schonger, E. Pulvermüller, and S. Sarstedt. Aspect Oriented Programming and Component Weaving: Using XML Representations of Abstract Syntax Trees. In *Workshop on Aspect-Oriented Software Development*, Bonn, Germany, 2002.