An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology

Michalis Anastasopoulos and Dirk Muthig

Fraunhofer Institute for Experimental Software Engineering (IESE)
D-67661 Kaiserslautern, Germany
{anastaso, muthig}@iese.fhg.de

Abstract. A systematic approach for implementing software product lines is more than just a selection of techniques. Its selection should be based on a systematic analysis of technical requirements and constraints, as well as of the types of variabilities, which occur in a particular application domain and are relevant for the planned product line (PL). In addition, each technique should provide a set of guidelines and criteria that support developers in applying the techniques in a systematic and unified way. This paper presents a case study that was performed to evaluate aspect-oriented programming (AOP) as a PL implementation technology. The systematical evaluation is organized along a general evaluation schema for PL implementation technologies.

1 Introduction

Software development today must meet various demands, such as reducing cost, effort, and time-to-market, increasing quality, handling complexity and product size, or satisfying the needs of individual customers. Therefore, many software organizations realize problems when developing and maintaining a set of separate software systems, and thus do or plan to migrate to a product line approach. A software product line is thereby a family of products designed to take advantage of their common aspects and predicted variability [1]. The members of a product line are typically systems in the same application domain; their independent development and maintenance usually require redundant and effort-intensive activities.

An approach for systematically developing and maintaining product lines is PuLSE^{TM} (Product Line Software Engineering)[1], which is used in technology transfer projects since 1998 [2]. The core of such an approach is the creation of a product line infrastructure that enables the efficient production of customer-specific products by explicitly managing their commonalities and variabilities.

While there has been significant effort spent in the product line community to systematize the early steps of product line engineering (PLE), that is scope definition, domain and feature modeling and architectural design, less attention has been paid to

[1] PuLSE is a trademark of the Fraunhofer Institute for Experimental Software Engineering (IESE)

the implementation level. There are, of course, many technologies available for implementing generic components but they are not well integrated with PLE. In other words, they target interested programmers but they lack in supporting organizations in systematically migrating to PLE. The latter could be achieved by collecting experience and knowledge in how implementation technologies performed in different contexts. The motivation for the PoLITe (Product Line Implementation Technologies) project [3] was to start the collection of experience and also to characterize and classify implementation technologies.

A systematic method for implementing software product lines is more than just a selection of techniques. For example, the selection must be based on a systematic analysis of technical requirements and constraints, as well as of the types of variabilities, which occur in a particular application domain and are relevant for the planned product line. In addition, each technique must provide a set of guidelines and criteria that support developers in applying the techniques in a systematic and unified way.

Conditional compilation [7], frames [8], template and generative programming [9] are examples of such techniques, as well as aspect-oriented programming (AOP) [10].

This paper presents a case study that was performed in order to evaluate AOP as a product line implementation technology. The study has contributed to the ViSEK portal [4], which aims at providing empirically validated knowledge in the general field of software engineering.

In section 2 a general evaluation schema for product line implementation technologies is presented. Section 3 then presents the case study applying AOP in the context of the larger GoPhone product line. Section 4 analyzes the results of the case study in combination with AOP and according to the schema presented before. Related work is discussed in section 0. Finally, section 6 concludes the paper by giving a brief outlook on future work planned.

2 Evaluation Schema

Software product lines pose special requirements on implementation technologies, which we divide into organizational and technical requirements. This paper, however, focuses on technical characteristics only. To technically evaluate product line implementation technologies, the contexts of both main activities in product line engineering, namely framework and application engineering, must be separately considered: the role of a technology while implementing a product line during framework engineering is significantly different from using a product line implementation while creating particular products during application engineering.

The following table provides an overview of the technical requirements for implementation technologies in terms of factors that influence framework and application engineering efforts. These factors are further analyzed in the subsequent paragraphs.

Table 1. Product line activities and associated requirements on implementation technologies

Activity		Effort	Factor
Framework Engineering	Implementing reusable code	Effort for making code reusable across the product line (development for reuse)	Reuse techniques
			Variation types
			Granularity levels
		Effort for testing reusable code	Testability
	Reacting to evolutionary change	Effort for integrating system-specific code into the product line	Component integration impact
		Effort for adding and removing variations (variability management)	Automation
		Maintenance effort	Reuse techniques
Application Engineering	Reusing code	Effort for reusing code to derive a concrete product (development with reuse)	Reuse techniques
	Resolving variations	Effort for creating a concrete product line member	Binding time
			Automation

2.1 Reuse techniques

Establishing strategic reuse is for most software product lines one of the main objectives. The difference between strategic and opportunistic reuse lies in the planning and realization of reusable artifacts with respect to anticipated reuse contexts in the future. The planning is thereby based on careful analyses of customer needs and technology trends. The goal is to identify features that should be made reusable across the whole product line, as well as features that are likely to arise as a result of software evolution.

2.1.1 Reuse across product line members

Flexibility is the major concern in development for reuse. The implementation technology must provide means for making code flexible so that it can reflect the various product requirements. This flexibility is usually achieved through generalization and decomposition. In [11] the two possibilities are called data-controlled variation and module replacement, respectively.

With generalization the goal is to create generic code that encloses many variations, while decomposition separates common from variant features so that on demand the latter can be attached to the common functionality.

Both approaches have well known advantages and disadvantages. Table 2 characterizes generalization and decomposition in terms of the effort that must be carried for framework and application engineering respectively.

Table 2. Characterizing generalization and decomposition

Approach	Activity	Effort for
Generalization	Framework Engineering	Making code abstract
	Application Engineering	Specializing code
Decomposition	Framework Engineering	Separating common from variant functionality
	Application Engineering	Integrating common and variant functionality

2.1.2 Reuse over time

Another reuse dimension is reuse over time [24], which affects both framework and application engineering. However, the latter refers to single-system evolution and is thus not considered here. Subsection 2.5 describes below issues arising when single-system evolution affects the product line. McGregor [12] distinguishes between proactive evolution techniques dealing with anticipated evolution, reactive techniques dealing with evolutionary change, and techniques automating framework configuration (see subsection 2.7 for more details).

Anticipated evolution is the only kind of evolution that enables a controlled management of activities since effort is spent for planning and identifying possible future changes and for preparing the system to accommodate the evolution. Nevertheless, it is practically impossible to foresee all future changes and predict the effort required for realizing them. Hence, unanticipated evolution cannot be fully avoided in practice. An ideal implementation technique thus supports both types of evolution.

2.2 Variation types

At the implementation level there are two main types of variability that have to be supported: positive and negative variability (these terms have been initially introduced in [13]).

In the first case functionality is added for creating a product line member while in the second case functionality is removed. The degree to which an implementation technique can handle different variation types is directly related to the effort required for development and reuse.

Negative variability is typically realized by generalization techniques. Generic components typically contain more functionality than actually needed in a specific product and for that reason functionality has to be removed. Positive variability is typically handled by decomposition techniques. In this case optional functionality is separated from the core and therefore it must be integrated, later, during application engineering.

Another issue in this context is the order in which variations are bound; an implementation technology should allow the programmer to define this order explicitly.

2.3 Granularity

Product line variability may exist at different levels of granularity ranging from entire components down to single lines of code. The levels of granularity that can be managed by a technology is an important characteristic of implementation approaches. If a product line architecture defines variation points that cannot be realized by a given implementation technology, restructuring the system may be necessary. For that reason, an implementation approach must be flexible and support various levels of granularity so that misalignments with variation defined in architecture or design is avoided.

When we use the Unified Modeling Language (UML) for describing product line architectures, its provided extension mechanisms enable a way of modeling variation points explicitly. For example, the stereotype <<variant>> may be attached to variant model elements representing variability as defined in [14]. This leads us to the identification of granularity levels shown in the following table. These levels can be further refined in adherence to the UML notation guide but this is beyond the scope of this paper.

Table 3. Granularity levels according to the UML

Classifiers	Relationships	Features	Procedures
Package	Generalization	Attributes	Read-Write Actions
Class	Association	Operations	Computation Actions
Interface	Dependency	Methods	Collection Actions
Data type			Messaging Actions
Component			Jump Actions
			Composite Actions
			• Group Actions
			• Loop Actions
			• Conditional Actions

2.4 Testability

Reuse and testing are tightly coupled because reusable code is thought of as quality-assured and this can be reached only through systematic testing. In a product line context there are special issues that have to be considered during testing [15].

As described in [16] code is testable when its behavior is observable and controllable at the code level. The extent to which these properties are reached depends on the implementation technology At this point the technology can be judged against the means it provides for creating testable as well as test code (i.e. code the performs tests).

2.5 Integration impact

During the evolution of a product line member, functionality is added, which at a point in time may become beneficial for other members as well. In this case the member-specific functionality must be integrated into the product line infrastructure. The same can happen with externally acquired components (e.g. COTS). The integration impact must be controllable, which poses an important requirement to the implementation technology used for creating the system-specific functionality.

2.6 Binding Time

Binding time refers to the point in time when decisions are bound for the variations, after which the behavior of the final software product is fully specified [18]. The binding time influences the effort for application engineering and may restrict the selection of the implementation technology [19]. The following figure illustrates typical binding times in the lifecycle of an application.

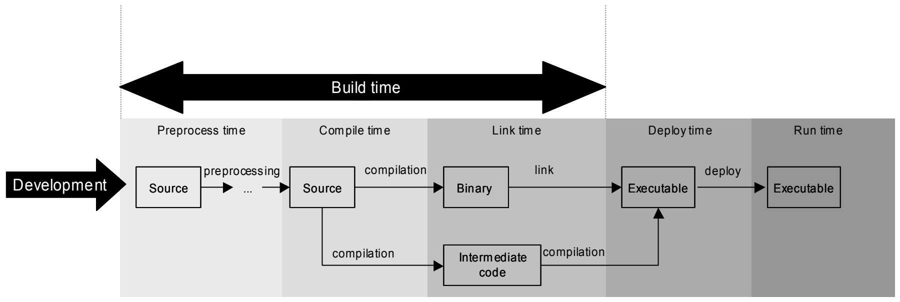


Fig. 1. Binding times in the life cycle of an application

2.7 Automation

Software product line engineering requires automation support both for framework and application engineering. Automation can reduce framework-engineering effort by automating the variability management. The latter includes adding and removing variations by taking compatibility and configuration knowledge [20] into account. During application engineering automation can also play an important role since it may simplify the creation of product line members by supporting the resolution of variation, for example, by decision models, which are models capturing the decisions to be taken while reusing generic product line artifacts.

The existence of decision models lies more in the responsibility of the general product line approach than of the implementation technology. The latter, however, also influences the overall achievable level of automation. Integrability of an implementation technology into a product line infrastructure is, therefore, also an important characteristic.

3 Case Study

The case study presented is based on a hypothetical mobile phone company, the GoPhone Inc. The study is meant to illustrate (a part of) a mobile phone software product line in a realistic way[2].

It happens often that features identified in the early steps of development cannot be directly mapped to units at the code level. This problem is also known as scattering [21] and becomes apparent when features are orthogonal to the rest of the system and so affect at the same time a large amount of other functionality. This happens typically with non-functional features. Aspect-oriented programming (AOP) has been conceived to match this situation.

AOP is a technology that has to be considered in product line engineering because it enhances, on the one hand, reuse of crosscutting features, and on the other hand, supports unanticipated evolution, which according to [24] is one of the fields where variability realization techniques fail to support evolution.

AOP distinguishes between components and aspects [22]: Components represent properties that can be encapsulated cleanly; aspects represent properties that cannot be encapsulated. Hence, aspects and components crosscut each other in a system's implementation.

Further concepts of AOP used in this paper are (for more details see [23]):

- Decomposition: At development-time, AOP decomposes an implementation into aspects or components. The aspect weaver is responsible for integrating aspects with components later.
- Development and Production Aspects: AOP is generally be used for two purposes. First, it "only" facilitates application development by handling functionality supporting development tasks like debugging, performance tuning or testing. Second, AOP is used to realize "real" functionality that is part of a released product as we did in the GoPhone case study (see below).
- Pointcut, advice and inter-type declarations: Pointcuts use type patterns for capturing points of concern (joinpoints) in the execution of a program. Advices contain the aspect code that is then introduced at these points. Finally inter-type declarations enable the modification of the static structure of a program.

In the case study, we selected the Java 2 Micro Edition (J2ME) as implementation technology despite the fact that, in practice, mobile phones are programmed in C. The motivation was to abstract from hardware-specific details and thus make the product-line-specific issues more visible and easier to understand. As a side effect, J2ME's phone emulators realistically visualize the running software.

The programming language of the GoPhone case study is Java and for that reason we decided to use AspectJ [23], which nicely integrates with development environments like Eclipse [25]. For Eclipse, it provides a plug-in with an aspect editor and views for managing crosscuts. Additionally, we have employed Ant [26] for automat-

[2] The documentation and source code of the case study is available at www.software-kompetenz.de.

ing the build process. Ant is an XML-based make tool, which is also well integrated with Eclipse.

3.1 Realizing the Optional T9 Feature

The input of text with a mobile phone is typically done in a rather uncomfortable way, namely with multi-tapping. This means that the user has to repeatedly press a button until he gets the right letter. T9 is an optional auto-completion feature that accelerates text input by predicting the words that the user wants to type [27].

The implementation of the optional T9 feature as an aspect has been realized by picking up all constructor calls to standard text fields, which enable text editing, and replacing them with constructor calls to an extended text field with T9 capabilities. The effort was considerably kept small by the fact that the T9 class inherits from the standard text field. So, extended methods are called automatically through polymorphism. Implementing a custom field that according to J2ME would not extend the text field class is also possible. In this case changing the static structure of the base through inter-type declarations, as well as advising at many additional pointcuts would be necessary.

In general, the implementation and management of the T9 feature and its optionality could be realized with the selected technology. The next section will analyze the case study in detail according to the schema presented above.

4 Analysis

4.1 Reuse Techniques

4.1.1 Reuse across product line members

AspectJ uses decomposition at development-time to separate components from aspects. However the decomposition is resolved during compilation, when aspect code is merged (woven) with component code.

The effort for application engineering, that is for integrating common and variant functionality is reduced to the minimum because the aspect weaver does this automatically. So, significant effort is found only during framework engineering.

The first step of isolating optional or alternative functionality into aspects is the identification of the variation points. This usually does not require much effort under the condition that the variability requirements are well understood and the design or code documentation is adequate. If this is not the case, aspect mining techniques (e.g. [17]) can help in that direction. The next step is to write the aspect code that will be introduced at the variation points, or in other words the joinpoints, found before. The effort required for this step depends on the granularity level and the type of the variation. As we will see later fine-grained variation inside of a method body is hard to deal with because AspectJ works basically at method and field level.

Another effort factor at this point is the question whether the base hierarchy can be maintained (see section 3.1 above). In other words if an aspect needs to break the class hierarchy at a joinpoint the effort may grow considerably.

4.1.2 Reuse over time

Aspect-oriented programming in general accommodates both types of evolution. Development for evolution (proactive) is mainly supported through the enhanced modularity, which comes with aspect-orientation. Ivar Jacobson [28] shows that the progressive change, which is inherent in every software development process, can be handled proactively with AOP. On the other hand AspectJ is an event-based mechanism as opposed to collaboration-based mechanisms that support preplanning by nature [29].

Reacting to evolution, as it arises, can be supported by the non-invasive composition, which is provided by AOP and particularly by AspectJ. For example, if a new crosscutting feature is required, developers add only the corresponding aspect code without changing the base system.

AOP can be seen as a transformation technique, which inherently is reactive [12]. Yet predicting the exact effects of AOP transformations can be a challenging task.

4.2 Variation types

AspectJ can support both types of variability although it is better suited for positive variability. Advice code is introduced before, after or instead of component code. Negative variability can be achieved at the method-level by replacing with less functionality. However AspectJ does not remove the original code from the resulting bytecodes. The original code will not be executed but it is included in the built product (it is even possible to call this code through reflection).

So, the suitability of AspectJ for negative variability depends on the reason why functionality must be removed. If for example some features must simply be disabled to provide a non-commercial version of a product AspectJ can be helpful. On the other hand if features have to be removed in order to run the product on a resource-constrained device other variability mechanisms should be considered.

As far as the order of variation binding is concerned, there are two ways that aspects can declare precedence. Either through the order in which aspect files are passed to the weaver or more effectively through special precedence declaration forms in the aspect code.

4.3 Granularity

AspectJ's basic primitive pointcuts are method and field-related. Attributes, Operations and Methods can be tailored easily and in a non-invasive way. AspectJ can alter the static structure of components by introducing new attributes and operations but it can also change the behavior of existing operations including read and write accesses to attributes.

Relationship, class, interface and data type variability can also be realized with AspectJ through inter-type declarations or by dynamically introducing new method calls in advice bodies.

Things get more complex with actions. In our case study we faced this problem especially with jump, loop and conditional actions. There is no pointcut in AspectJ that can capture a variation at this level. The same applies to loop and conditional actions. There is no way we can tailor the expressions that control an iteration or a conditional block. For such kind of variations other techniques like Frames are better suited [30].

4.4 Testability

Testability is one of the open issues with AOP [31]. Aspect code always depends on the component code and is therefore difficult to test in isolation. Moreover, AspectJ weaves at the binary level and thus makes the source of the woven code unavailable. Thus structural testing of woven code becomes difficult. To this comes the problem of aspect correctness. Making sure that an aspect will work correctly upon reuse is another open issue [32].

Although code residing in aspects may be not easy to test, it can be used for testing existing code. Development aspects are a good way for assuring contract enforcement and tracing program execution or for injecting faults into a system and observing its reaction. As much as product line testing is concerned, the creation of generic test cases based on AOP is conceivable.

4.5 Integration impact

The impact of integrating aspect code into an existing system depends on the type patterns used. In other word pointcuts or inter-type declarations that use wildcards impose a great impact. AspectJ provides however tools that show the affected components and so enable a better overview and control of the impact.

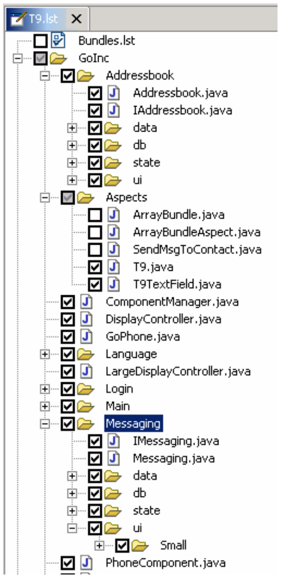
Another problem with the integration of aspect code lies in the conflicts that can occur. This happens especially when inter-type declarations come into play. An aspect cannot declare a public inter-type member that already exists as a local class member.

4.6 Binding time

AspectJ weaves code at the bytecode level. That means that only the aspect code needs to be compiled upon weaving. AspectJ compiles and merges the aspect code directly with the existing binaries. However the component code must be recompiled when woven aspect code must be removed (negative variability).

4.7 Automation

In this work we used the AspectJ and Ant plug-ins for Eclipse to support the decision making process when deriving products. For configuring a product, the AspectJ plug-in provides an editor, which enables defining different configurations by simply checking the boxes of the modules required. Per default, the editor unfortunately provides check boxes for all files (see Fig. 2). For that reason we employed Ant towards a more effective way for packaging configurations.

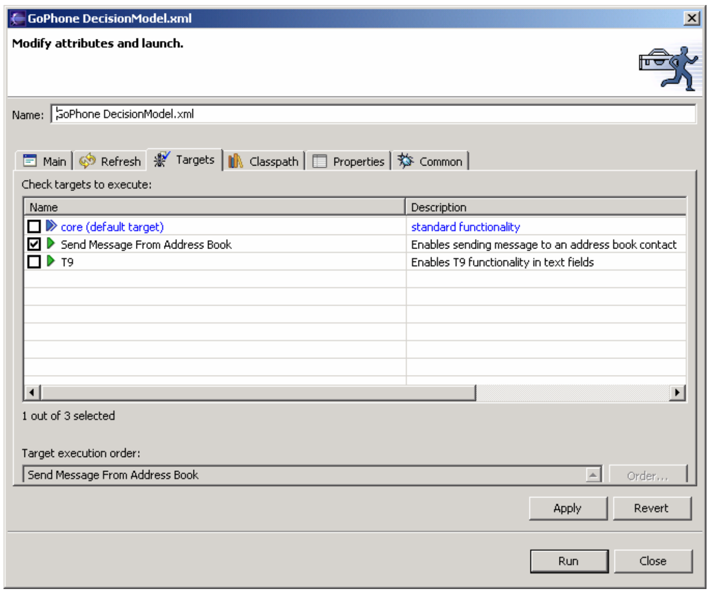


```
T9.lst ×
☐ Bundles.lst
☑ GoInc
    ☑ Addressbook
        ☑ Addressbook.java
        ☑ IAddressbook.java
        ☑ data
        ☑ db
        ☑ state
        ☑ ui
    ☑ Aspects
        ☐ ArrayBundle.java
        ☐ ArrayBundleAspect.java
        ☐ SendMsgToContact.java
        ☑ T9.java
        ☑ T9TextField.java
    ☑ ComponentManager.java
    ☑ DisplayController.java
    ☑ GoPhone.java
    ☑ Language
    ☑ LargeDisplayController.java
    ☑ Login
    ☑ Main
    ☑ Messaging
        ☑ IMessaging.java
        ☑ Messaging.java
        ☑ data
        ☑ db
        ☑ state
        ☑ ui
            ☑ Small
    ☑ PhoneComponent.java
```

Fig. 2. T9 Build Configuration (excerpt)

The targets shown in the following picture reflect different product line features. The core target represents the base component code while the other two targets represent optional features. The Ant script behind this dialog box interacts with the AspectJ compiler in order to build a product containing the selected features.

Adding a new feature means therefore adding a new target in the Ant script. The according effort is kept small because the only difference from existing targets lies in the different aspect files that will be given to the AspectJ compiler. That means that a target template can be reused each time a new aspect is added.



GoPhone DecisionModel.xml

Modify attributes and launch.

Name: GoPhone DecisionModel.xml

Check targets to execute:

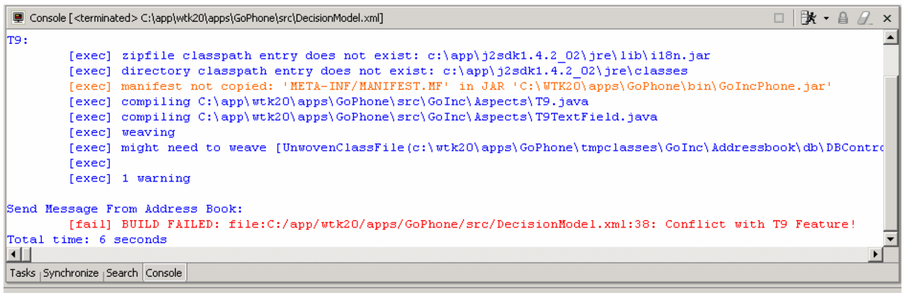
	Name	Description
☐	core (default target)	standard functionality
☑	Send Message From Address Book	Enables sending message to an address book contact
☐	T9	Enables T9 functionality in text fields

1 out of 3 selected

Target execution order:
Send Message From Address Book

Fig. 3. Automating the decision making process

Ant can also be used to support conditions as well as dependencies between features. The following picture shows the output of the build process when an invalid feature combination has been selected. As shown in the picture, the optional functionality is being compiled and then woven with the already compiled core.



Console [<terminated> C:\app\wtk20\apps\GoPhone\src\DecisionModel.xml]

```
T9:
        [exec] zipfile classpath entry does not exist: c:\app\j2sdk1.4.2_02\jre\lib\i18n.jar
        [exec] directory classpath entry does not exist: c:\app\j2sdk1.4.2_02\jre\classes
        [exec] manifest not copied: 'META-INF/MANIFEST.MF' in JAR 'C:\WTK20\apps\GoPhone\bin\GoIncPhone.jar'
        [exec] compiling C:\app\wtk20\apps\GoPhone\src\GoInc\Aspects\T9.java
        [exec] compiling C:\app\wtk20\apps\GoPhone\src\GoInc\Aspects\T9TextField.java
        [exec] weaving
        [exec] might need to weave [UnwovenClassFile(c:\wtk20\apps\GoPhone\tmpclasses\GoInc\Addressbook\db\DBContro
        [exec]
        [exec] 1 warning

Send Message From Address Book:
        [fail] BUILD FAILED: file:C:/app/wtk20/apps/GoPhone/src/DecisionModel.xml:38: Conflict with T9 Feature!
Total time: 6 seconds
```

Fig. 4. Supporting Feature Dependencies

At this point it must be noted that this kind of automation is a simple script-based solution and for complex scenarios probably not sufficient. It can however be combined with sophisticated configuration management environments like GEARS [36].

4.8 Summary

The following table summarizes the evaluation results according to the schema of Table 1.

Table 4. Evaluation summary

Activity		Effort	AspectJ
Framework Engineering	Implementing reusable code	Effort for making code reusable across the product line (development for reuse)	Effort increases with: ▪ Fine granularity ▪ Break of base hierarchy
		Effort for testing reusable code	Aspect testability is a well-known drawback
	Reacting to evolutionary change	Effort for integrating system-specific code into the product line	Impact increases with: ▪ Type pattern genericity ▪ Introduction assumptions
		Effort for adding and removing variations (variability management)	Eclipse integration contributes to automation support
		Maintenance effort	Less effort in the maintenance of crosscutting features
Application Engineering	Reusing code	Effort for reusing code to derive a concrete product (development with reuse)	Effort is normally kept small
	Resolving variations	Effort for creating a concrete product line member	▪ Less effort for compile-time binding ▪ Decision support through configuration builder or Ant-based automation

5 Related Work

The evaluation of product line implementation technologies has received attention in the research community [19, 24, 34]. Yet it is not clear how different technology dimensions such as configuration management, component-orientation and programming language techniques can be combined for efficiently implementing product lines. This constitutes the major goal of the PoLITe project [3].

On the other hand the specific selection and evaluation of AOP as a product line implementation technology has been the focus of a few papers:

Lopez-Herrejon and Don Batory [33] used AspectJ to implement a simple family of graph applications. The authors face the problem of illegal feature compositions and discuss that AspectJ does not support the developer in this direction. Indeed, AspectJ does not provide any explicit support, except of its aspect prioritization feature that may help in this situation. However as shown in section 4.7 we can define such rules outside AspectJ, for example in ant scripts. Moreover, this issue can be approached with special-purpose aspect code [35].

The next problem encountered by the authors relates to the management of the files passed to the weaver. In the graph product line all features are aspects and thus deciding which ones are passed to the weaver is an error-prone task. Yet this issue can be handled with ant scripts, as we demonstrated in section 4.7, or with more sophisticated tools like GEARS [36].

Beuche and Spinczyk [37] demonstrate how AspectC++ (the counterpart of AspectJ for C++) can be employed in a weather station product line. The use of AspectC++ simplified the development process since crosscutting product line features could be directly mapped to aspects. Moreover, a major issue of the weather station product line was the decision of the level, at which hardware interrupts are controlled. Aspect C++ enabled configuring the appropriate level and therefore enhanced the code reusability for various application needs.

Finally, Sven Macke in his diploma thesis [38] uses AspectJ as a generative technique for the implementation of a search engine. One of the major problems encountered was the restriction that aspect code can only be woven with component code that resides in source files. The author missed the possibility of weaving with code residing in a database as a stored procedure. This led to the important conclusion that configuration knowledge and therefore automation support cannot be fully handled by using only AspectJ.

6 Conclusion

The case study has shown that AOP is especially suitable for variability across several components. Whether AOP may be suitable for variability of different kind and inside of single components will be subject of future work. It will also explore the question whether alternative implementation technologies (e.g. frames) are available that handle some variability better.

In parallel the integration of AOP into general product line approaches should be improved to further enhance the achievable automation level.

7 Acknowledgments

We would like to thank the members of the Software Product Line (SPL) department at Fraunhofer IESE for their comments and feedback, especially Joachim Bayer, Martin Becker, Thomas Forster and Thomas Patzke.

8 References

1. David Weiss, Chi Tau Robert Lai, Software Product-Line Engineering. A Family-Based Software Development Process, Addison-Wesley, 1999
2. Homepage of PuLSE^{TM}, http://www.iese.fhg.de/pulse/

3. Homepage of the PoLITe project, http://www.polite-project.de/
4. Homepage of the ViSEK project, http://www.visek.de/ (in German)
5. Dirk Muthig, Michalis Anastasopoulos, Roland Laqua, Stefan Kettemann Thomas Patzke, Technology Dimensions of Product Line Implementation Approaches State-of-the-art and State-of-the-practice Survey, IESE-Report No. 051.02/E, September 2002
6. P. Cointe(Ed.). ECOOP '96: Object-oriented programming. Proceedings of a Workshop held at the 10th European Conference on Object-Oriented Programming, Linz, Austria, 1996
7. B. Stroustrup, The C++ programming language, third edition, Addison-Wesley, 1997
8. Paul Basset, Framing Software Reuse. Lessons From the Real World, Yourdon Press, 1997
9. Krzysztof Czarnecki, Ulrich Eisenecker, Generative Programming. Methods, Tools and Applications, Addison-Wesley, 2000
10. Tzilla Elrad, Robert E. Filman and Atef Bader, Aspect-oriented Programming, COMMUNICATIONS OF THE ACM October 2001/Vol. 44, No. 10
11. Felix Bachmann, Len Bass, Managing Variability in Software Architectures, In proceedings of the Symposium on Software Reusability, SSR'01, May 18-20, 2001, Toronto, Ontario, Canada.
12. John D. McGregor, The Evolution of Product Line Assets, Report Number CMU/SEI-2003-TR-005, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, June 2003
13. J. O. Coplien: Multi-Paradigm Design for C++. Addison-Wesley, 1999
14. Colin Atkinson et al., Component-based Product Line Engineering with UML, Component Software Series, Addison-Wesley, 2001
15. Ronny Kolb, Dirk Muthig, Challenges in Testing Software Product Lines, CONQUEST'2003. 7th Conference on Quality Engineering in Software Technology - Proceedings (2003), 103-113 : Ill.
16. John D. McGregor, Testing a Software Product Line, Report Number CMU/SEI-2001-TR-022, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, December 2001
17. J. Hannemann and G. Kiczales, Overcoming the prevalent decomposition of legacy code. In Proc. of Workshop on Advanced Separation of Concerns at the International Conference on Software Engineering (ICSE), Toronto, Canada, 2001.
18. Charles Krueger, Towards a Taxonomy for Software Product Lines, in Proceedings of the 5th International Workshop on Product Family Engineering. Siena, Italy. November 2003.
19. Claudia Fritsch, Andreas Lehn, Dr. Thomas Strohm, Evaluating Variability Implementation Mechanisms, in Proceedings of International Workshop on Product Line Engineering, Seatle, USA, 2002
20. Krzysztof Czarnecki, Ulrich W. Eisenecker, Generative Programming, Methods, Tools, and Applications. Addison-Wesley, 2000
21. Peri Tarr, Harold Ossher, William Harrison, Stanley Sutton. "N Degrees of Separation: Multi-Dimensional Separation of Concerns." ICSE 1999 Conference Proceedings. pp 107-119, 1999
22. Gregor Kiczales et al, Aspect-Oriented Programming, Published in proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.
23. The AspectJ Programming Guide, the AspectJ team, Xerox Corporation, 2002-2003 Palo Alto Research Center
24. Mikael Svahnberg, Supporting Software Architecture Evolution, Blekinge Institute of Technology Doctoral Dissertation Series No 2003:03
25. Homepage of the Eclipse project, http://www.eclipse.org/
26. Homepage of Ant, http://ant.apache.org/

27. Homepage of T9, http://www.t9.com/
28. Ivar Jacobson, Use Cases and Aspects–Working Seamlessly Together, IBM Corporation, 2003, available at http://www.ivarjacobson.com
29. Don Batory, Jia Liu, Jacob Neal Sarvela, Refinements and Multi-Dimensional Separation of Concerns, in Proceedings of ESEC/FSE'03, September 1-5, 2003, Helsinki, Finland
30. Thomas Patzke, Dirk Muthig, Product Line Implementation with Frame Technology: A Case Study, IESE-Report No. 018.03/E, March 2003
31. Roger Alexander, The Real Costs of Aspect-Oriented Programming, Quality Time, IEEE Software, November/December 2003
32. Tzilla Elrad, Moderator, Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, and HaroldOssher, Panelists, Discussing Aspects of AOP, Communications of the ACM, October 2001/Vol. 44, No. 10
33. Roberto E. Lopez-Herrejon, Don Batory, Using AspectJ to implement product lines, A case study, Technical Reports 2002, Department of Computer Sciences, The University of Texas at Austin, available at http://www.cs.utexas.edu
34. Roberto E. Lopez-Herrejon and Don Batory, A Standard Problem for Evaluating Product-Line Methodologies, in Third International Conference on Generative and Component-Based Software Engineering, September 2001, Erfurt, Germany.
35. Ramnivas Laddad, AspectJ in Action, Manning Publications, June 2003
36. Dr. Charles W. Krueger, Software Mass Customization, October 2001, BigLever Software Inc, available at http://www.biglever.com
37. Danilo Beuche, Olaf Spinczyk, Variant Management for Embedded Software Product Lines with Pure::Consul and AspectC+, in Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, October 2003, Anaheim, CA, USA
38. Sven Macke, “Generative Programmierung mit AspectJ“ (in german), Diploma Thesis, Fachhochschule Kaiserslautern, Angewandte Informatik, 2001