Applying Aspect-Orientation Techniques in Automotive Software Product-Line Engineering

Katharina Mehner
Technische Universität Berlin
Fachgebiet Softwaretechnik
Sekretariat FR5-6
Franklinstraße 28/29
10587 Berlin, Germany
mehner@cs.tu-berlin.de

Mark-Oliver Reiser
Technische Universität Berlin
Fachgebiet Softwaretechnik
Sekretariat FR5-6
Franklinstraße 28/29
10587 Berlin, Germany
moreiser@cs.tu-berlin.de

Matthias Weber
DaimlerChrysler AG
Research & Technology
Alt-Moabit 96a
10559 Berlin, Germany
Matthias.N.Weber@
DaimlerChrysler.com

Abstract

The application of product-line engineering concepts to automotive software development has received increasing attention during the last few years. A core facet of software product-line approaches is the definition and management of variability within various development artifacts. Aspect-orientation techniques provide an interesting alternative to the traditional way of defining variability with variation points and variants.

In this paper we (1) show where and how aspect-orientation can be embedded in an automotive product-line approach, (2) compare the traditional technique for variability definition with the aspect-oriented one and finally (3) discuss the benefits as well as possible shortcomings of applying aspect-orientation for automotive software product-lines.

1. Introduction

One of the challenges in today's automotive development is the high degree of variability of the developed systems. Manufacturers sell their products in many countries with diverse customer expectations, local legislation, etc. Moreover, being able to offer a broad range of products appealing to very different target groups became, over the last years, an important prerequisite to a competitive position on the global markets. On the other hand, suppliers sell their products to different manufacturers, thus having to tailor their products to the needs of each customer.

A promising approach to handle this complex situation is product-line engineering (cf. [PBL05]), which became quite popular during the last few years. At the heart of a product-line framework is the definition of the variability in each development artifact (e.g. requirements databases, design models, test cases). Traditionally this is achieved by specifying variation points in the variable artifacts together with several variants for each variation point. Unfortunately this approach has severe weaknesses. For example, often a single source of variation affects development artifacts at many different locations, spreading its definition across many variation points all over the artifact. In addition, keeping apart different sources of variation – i.e. different optional features of the system – is impossible when they affect a same variation point within an artifact, leading to highly intertwined variation definitions. This results in extremely complex definitions for variable artifacts which are error prone and difficult to maintain and evolve.

Aspect-oriented techniques are an interesting alternative to this traditional style of defining variability and may help overcome its specific problems. Below, we will describe what role aspect-orientation may play within the context of an automotive software product-line, compare it with the traditional approach and discuss benefits and possible problems of this new paradigm with respect to automotive development.

The remainder of this paper is structured as follows: In the next section, we first give an overview of product-line oriented automotive software engineering as the methodological context of this discussion. The basic idea of aspect-orientation is described in Section 3. In the subsequent two sections we explain the traditional (Section 4) and the aspect-oriented way (Section 5) of defining variability in software artifacts. These two approaches and their impact on and value

International Automotive Requirements Engineering Workshop (AuRE'06)
0-7695-2710-8/06 $20.00 © 2006 IEEE

for automotive development are then compared to each other in Section 6. The text concludes with a discussion of related work and ideas for further research.

2. Software product-lines

In this section we give an overview of product-line oriented software development and show at what point within this context aspect-orientation techniques may be applied. This will also lead to several requirements on the concepts introduced in Section 5.

The basic idea of product-line oriented software development is that instead of developing several similar but still significantly distinct software products independently from each other, only a single, but variable product is developed and – in a second step – the individual products needed are derived from the variable product by configuration. The variable product is often referred to as the *product-line infrastructure* whereas the individual products are called *product instances*. For example, the climate control system for Mercedes-Benz A-Class, C-Class and S-Class could be developed independently from another, i.e. each phase of software development – such as requirements analysis, design, implementation, integration and test – is performed for each type of climate system separately. In contrast, following a product-line oriented approach, only one variable climate control system would be developed which would then be configured to meet the needs for A-Class, C-Class and so on.

It is important to note that 'configuration' is used in a very broad sense here:

- It could mean that the binary code of the climate control actually running on a shipped vehicle is the same for A-Class, C-Class and S-Class vehicles and that this identical code is only configured through parameters in flash memory.

- It could as well mean that already the behavioral description (e.g. a Statemate diagram) is configured and that code generation is performed on such a configured model, thus leading to shipped code that is different for the various vehicle types, only containing the functionality needed in the vehicle it is deployed in.

These are only two of a multitude of possible ways of *resolving variability*. In a complex system, usually more than one of these are applied simultaneously.

In summary, using a product-line approach means that all artifacts of the software development process may be variable. For example, a requirements document for the climate control may contain requirements that are only needed for S-Class or that change in meaning depending on the targeted type of vehicle. The same applies to design models (e.g. component diagrams), behavioral descriptions, test cases etc. This is illustrated in the lower half of Figure 1. The empty rectangle in the enlarged artifact on the right side indicates a *variation point*, i.e. a point where the content of the artifact is variable. The three rectangles further to the right denote *variants*, i.e. different content that may be included in the artifact at the variation point.

When taking into account all variable artifacts, this leads to a multitude of variation points, variants and possible configurations, creating an enormous complexity. To manage this, many product-line approaches use *feature models*. In this context, a *feature* is a characteristic or a trait a specific product instance of the product-line may or may not have. For example, a feature of the climate control may be "Low Energy Consumption" for vehicle types with a limited on-board power supply or for markets with harsh requirements on fuel consumption. A feature model lists all these features together with dependencies between them.

In contrast to requirements, features are not intended to list *all* characteristics of a given product instance but only those that are needed to tell apart the product instances from one another. However, features and requirements share many similarities and features may well be seen as coarse-grained requirements.

There is no received opinion in product-line literature of precisely how to apply feature models in the development process. We propose to have a central, global feature model for all development artifacts (see upper half of Figure 1), used by different departments – development, production, sales, marketing, management – and that serves (1) as a common basis for configuring all artifacts, (2) as a means to organize and maintain the variability of all artifacts and (3) as a set of coarse-grained requirements.

In order to use the central feature model as a basis for configuration, the engineer has to define for each variant when it will be selected in terms of selection and deselection of the features in the central feature model (*variant selection definition*). Then, a configuration of the central feature model – i.e. a selection or deselection of its features – can be used to configure the development artifacts.

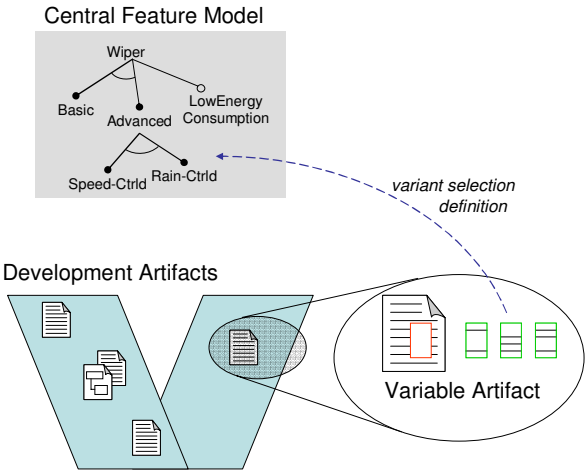

Fig. 1. Product-line oriented development.

The various product-line approaches proposed by the scientific community differ quite substantially with respect to details. For example, an important detail is how to precisely define the variability of an artifact, i.e. how to define the existing variation points, the variants for each variation point and their selection definition. The answer to this question of *artifact variability definition* is highly dependant on the type of artifact you are looking at: A variation point in a requirements document has to be described in a different fashion than one in a component diagram or a state diagram. And since the engineer defining and maintaining the artifacts is closely working with this concept and since the complexity can be very high (large number of variation points, etc.), the usability and scalability of this concept is of utmost importance.

At this point aspect-orientation comes in. In the remainder of this paper we will examine how aspect-orientation could be applied as a valuable technique for artifact variability definition to overcome the shortcomings of the traditional approach as described in Section 1.

3. Aspect-orientation

Let us now introduce the basic idea of aspect-orientation before, in later sections, looking in more detail at the definition of artifact variability both in a traditional and an aspect-oriented way.

Aspect-orientation (AO) aims at improving the modularization for crosscutting concerns, i.e., for concerns that are found in many modules of a given modularization and hence are not well localized. Aspect-orientation provides a novel module concept called *aspect* for encapsulating the structure and behavior of a crosscutting concern and a novel composition paradigm that composes aspect behavior with existing modularization through method call interception.

The idea of aspect-orientation was first applied to programming languages as *aspect-oriented programming* (AOP). Therefore it is useful to start with examining an example in this area. Consider the following code snippet:

```
class Client {

   private String name;
   private int    age;
   ...
   void setName(String newName) {
      name=newName;
   }
   String getName() {
      return name;
   }
   void setAge(int newAge) {
      age=newAge;
   }
   int getAge() {
      return age;
   }
}
```

Now, if we needed to add some functionality – e.g. error handling or logging for debug purposes – to all methods of class Client that are setting values within this class, we would have to add appropriate code in methods setName() and setAge(). The drawback of this would be, for example, that when adding a new attribute together with a new setter-method, we would have to remember to add this functionality also to this new method. Furthermore, the code needed to be added to the various methods will be very similar or even identical in many cases, leading to redundant code with its typical shortcomings.

The solution that aspect-oriented programming has to offer here is the definition of an *aspect*. An example of such an aspect, specified informally in natural language, might be:

```
Whenever
   a method of class Client with a name matching
   the pattern "set*" is called
execute the following code:
   System.out.println("DEBUG-INFO: „+
      „Client attribute changed!");
```

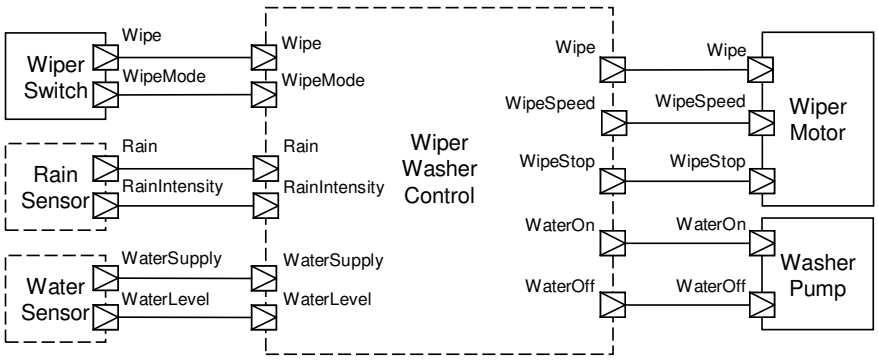

Fig. 2. WiperWasher Component Diagram.

This simple example already shows the main constituents of an aspect: A definition *where* some additional functionality is to be executed in the running program – called a *point-cut* in AO terminology – and a definition of *what* functionality is to be inserted – called an *advice*. The process of inserting advices into a running program is usually referred to as *aspect weaving*. Aspect-oriented programming languages, such as AspectJ [Mil04] or ObjectTeams [Her02], provide means to formally specify the point-cut and the advice. Note that the definition of a formal language to express point-cuts also implicitly defines at what points of a program advices may theoretically be woven in. The set of these theoretically possible targets of aspect weaving are called the *join-points* of a program or, speaking on a more general level, of a programming language.

Today's aspect-oriented programming languages provide a tremendous degree of flexibility in defining point-cuts and advices. For example, it is possible within an advice to reflect on the context in which it is actually woven in. In the simple aspect shown above, this would allow us to output the name of the changed attribute together with its old and new value. In the point-cut, on the other hand, many more characteristics of the target code than only method names can be referred to in order to define where the aspect is woven in. For example, only methods with a certain return type may be chosen or only methods with a first parameter of type Integer. In addition to these static characteristics, *dynamic aspects* allow to also refer to dynamic properties of the running code, in order to define whether or not an advice is woven in. The question of "where" weaving takes place is altered to "where and when" weaving takes place. With this facility, we could formulate an aspect, that prints an error message whenever a Sting-attribute is about to be set to 'null':

Whenever
a method of class Client with a name matching the pattern "set" and a single parameter of type String is called,* (static characteristics so far)
AND this first parameter is 'null' (dynamic)
execute the following code:

```
System.out.println("ERROR: "+
   "Attribute "+attribName+
   " set to null !");
```

There are many more possibilities with respect to aspects: for example, advices can also be executed after a method finishes or advices can change the values of parameters or return values. Further details of these point-cut and join-point definitions are beyond the scope of this overview. Instead, for our discussion, it is more important to concentrate on the basic idea behind aspect-orientation. This can be summarized as follows:

1. an *aspect* consists of a point-cut and an advice
2. the *point-cut* specifies where (or where and when) the advice will be executed
3. the *advice* specifies what is executed (program code in our example)
4. the formalism used to define point-cuts implicitly defines a set of possible points of insertion, called *join-points*

At this point it is important to note that the basic artifact which is changed by weaving in aspects does not need to be changed in any way, i.e. there is no need to explicitly mark the join-points in the code. Instead they are implicitly defined as above. We will further discuss this property of the aspect-oriented approach below.

The kind of point-cuts we have introduced with our examples support dynamic crosscutting [BCC05], because the point-cuts match points in the execution of a program and thus will have an effect on the runtime

0-7695-2710-8/06 $20.00 © 2006 IEEE

behavior. There exists also static crosscutting which supports the manipulation of the structure of a program, e.g., introducing members to classes.

4. Traditional variability definition

In this section we will first examine how artifact variability is traditionally defined with variation points and variants before we explore the potential of using aspect-orientation for this purpose in Section 5.

We illustrate variability by means of a wiper control example using a hierarchical component notation called EAST-ADL, a modeling language specifically targeted at the automotive domain[1]. The example describes the high-level software components required to realize the control of a wiper and washer system. The wiper switch is used to trigger the system. The two sensors provide additional information. From all these inputs, signals are generated that control the actual wiper motor and washer pump (see Figure 2).

Components are connected through their in-ports and out-ports, which are denoted by special graphical icons with a triangle pointing inward or outward. Here we cannot give more details on the communication semantics between components. A component can be refined by a set of other components which is omitted for reasons of simplicity.

Each component can be a variation point. This means that such a component is merely a place holder but has no specification on its own. Such a component is depicted with a dashed outline. The variants for a variation point are specified in a separate diagram (see Figure 3). A variant can specify all the ports of the variation point but does not have to.

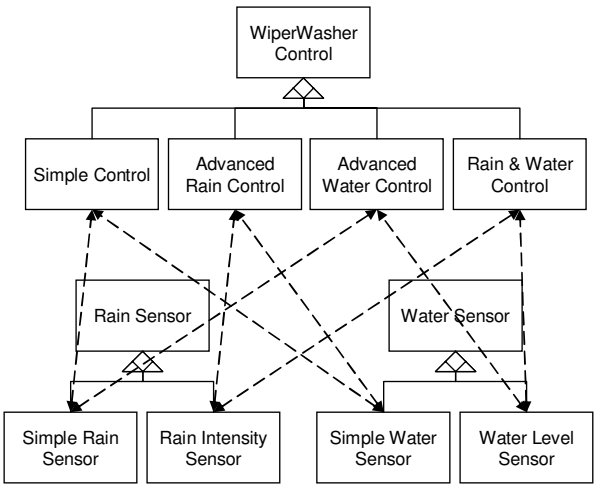

Fig. 3. WiperWasher Component Variability and Dependencies.

[1] The details of this notation are beyond the scope of this paper, but the concepts we use here are very similar to UML2 component diagrams.

In the example, the two sensors' components and the control component are variation points. For each sensor, a simple and an advanced version exist. The simple rain sensor signals only that it is raining, while the advanced sensor also signals the intensity. The simple water sensor signals whether water for washing is available or not while the advanced sensor also signals the water level. The level can be used for implementing a water-saving mode when the water container is almost empty. Depending on the chosen sensors different controls have to be used. There is one control for each combination of sensors. Dependencies between variants are depicted as a mutual dependence relationship using dashed arrows.

As already mentioned, the selection of appropriate sets of variants is controlled through feature modeling. For each variant, it can thus be defined on which features it depends.

In the remainder of this section, we will discuss the potential for improvement in the notation used for variability. The notation can be evaluated under the perspective of understandability and flexibility.

1. The variants are grouped according to which component they replace. It is often the case that dependencies arise between variants. This additional dimension cannot be captured and is therefore difficult to understand. The information belonging to one configuration is scattered over the individual variant definitions. Especially, if there are many dependencies, the workaround of annotating them in the variant diagram does not scale. Conversely, in the variant notation, different variations become tangled with each other. The problem causing this situation is also known as the tyranny of the dominant decomposition, which is here the component definition. According to the component definition language, the variant definition language has been defined such that a single component is a variation point and a single component is a variant.
2. The notation requires specifying the variation points precisely, and thereby, determining at what level of abstraction they are introduced. If, however, a more fine grained variation point deeper in the hierarchical refinement of a component would be sufficient, nevertheless, the entire context as specified through the variation point has to be included. If, on the other hand, a more coarse

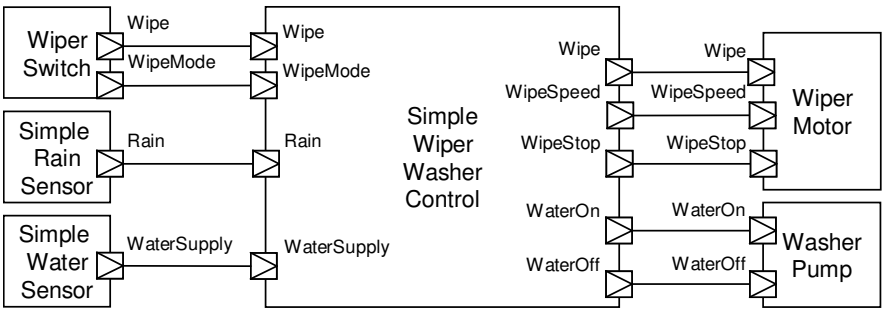

Fig. 4. WiperWasher Component Diagram for Default.

grained variation point would be needed, instead, several variation points have to be used to achieve the variation in a larger context. Both workarounds hamper especially adaptation and evolution of specifications and the underlying systems.

5. Variability definition with aspects

In this section, we propose an alternative for modeling variability which follows the ideas of aspect-orientation. While aspect-orientation primarily aims at improving modularization for a concrete model, it is also conceivable for using it for the additional dimension of variability. Here, it should try to overcome the problems pointed out at the end of the last section.

We aim at replacing the notion of variation points and also the notion of a variant diagram by means of aspects.

1. The notion of a point-cut lends itself naturally as a description of a variation point. Not only individual components can be referred to in a point-cut but also a set of components.
2. The part corresponding to an advice will specify the variants which can be a single component but also a set of components. Moreover, the advice will capture all variants that are dependent on each other and belong to a configuration.

As we aim at not determining before hand, where the variation points are, there is one important implication. Now, all the components in the normal diagrams must have a specification, i.e. there are no more place holders. Thus, a component diagram specifies a default, basic configuration (see Figure 4).

The new concepts are illustrated with our running example. Instead of the diagram with place holders we now have a base diagram showing a configuration using two simple sensor components and a simple control component. There are three additional configurations. Two of them exchange one of the sensors, and one configuration replaces both sensors by advanced sensors. Each configuration also exchanges the control component.

This is modeled as follows. The point-cut of an aspect consists of a graphical representation of the component that should be replaced. This is depicted in the upper half of a grey box, separated by an arrow from the lower half which depicts the new configuration (see Figure 5).

While in the traditional variability model, the place holder already contained the maximum number of ports, in this model the default configuration only depicts the ports needed in this configuration. A new configuration can add additional ports when needed.

The idea of completely replacing components for other variants is a reminiscent of the traditional variation-point / variant approach. This might even include the deletion of a component in order to simulate the null variant option. Using aspect-oriented techniques, it is, however, also conceivable that a component is re-used in a variant configuration.

In the example, instead of replacing the simple control when an advanced sensor is used, the simple control could be adapted by another component by pre- or post-processing its inputs / outputs. This is an alternative to the aspects of Figure 5. It is exemplarily described for the advanced rain sensor in Figure 6. Here, we have shown explicitly all ports, to which the adaptor control can connect in order to adapt the existing control. In the examples, there are components that are matched by the point-cut but that are not changed by the aspect. Here, they have to be repeated in the advice. This redundancy could be avoided by designing a more appropriate language for defining point-cuts and aspects.

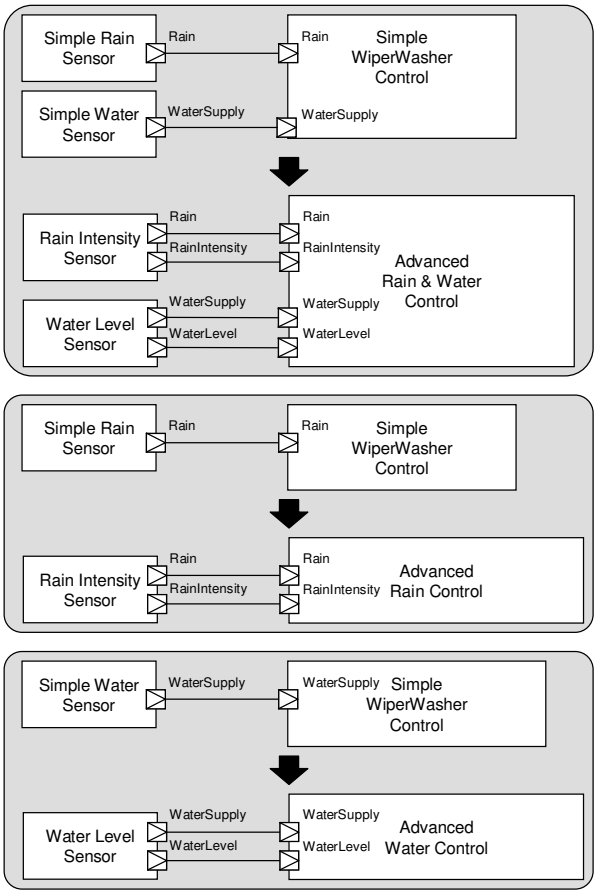

Fig. 5. WiperWasher Aspect Diagram.

As mentioned before, the selection of an aspect in our approach is additionally triggered by the selection of features from the feature models. This is not shown in the example but it should be pointed out that this is not present in standard aspect-orientation, where the aspect is applied when the point-cut finds a match.

Also, here we have not yet introduced point-cuts which specify properties of components they are intended to match but only concrete components. It is however conceivable to work with matching as this ideally captures problems that are crosscutting. Point-cuts could refer to characteristic patterns in the internal structure of a component or in the context of a component and hence match more than one component. Crosscutting can be found e.g. in redundancy or plausibility analysis of system control.

Note that here, we have only considered components as join-points. It is also conceivable to use ports and port-connections as join-points. This is a topic of future research.

6. Comparison

Now, we will compare the two approaches for defining the variability of a product-line's artifacts, namely the traditional variation-point/variant approach outlined in Section 4 and the alternative approach using aspect-orientation presented in Section 5. We will first look at similarities and differences on a technical level and then, in a second step, examine the implications of these differences for automotive software development from a methodological point of view.

6.1. Technical differences

The most important difference between the two forms of variability specification is related to the question where variability occurs within an artifact: Traditionally this is specified by adding some kind of variation point element in the variable artifact's description (as shown in Section 4 for component diagrams). In contrast, the AO approach does not alter the base diagram. Instead, the point where a variation occurs is defined only externally in the point-cut of an aspect (again, shown for component diagrams in Section 5). So, on the one hand, variation points are specified explicitly by way of dedicated elements, on the other hand, variation points are specified only implicitly, offering the possibility of an unaltered base model.

Another important technical difference is the relation between the points where variation occurs and the definition of the actual variation. The traditional approach of using variation points and variants defines the variation for each variation point separately. In other terms, each variant is related to exactly one variation point. This tight coupling of variation definition and the points where the variation is about to take place is detrimental when one and the same variation should occur at many different points "scattered" all over a complex model. For example, let us assume that for a complex diagram of an entire body electronics system (with control for wiper, headlights, blinker, windows, climate, etc.) we want to add optional diagnostics functionality to all sensors that allows the logging of the sensors' data over some diagnostics interface. Even if the functionality is identical for each sensor, we would need to specify a variation point together with appropriate, dedicated variants for each sensor. Aspects, on the other hand, would allow us to define the added functionality once (in form an advice) an insert this at each sensor by formulating a point-cut that selects all sensors in much the same way as we added logging functionality to several java methods in Section 3. In AO terminology: The aspect allows us to weave in the *cross-cutting concern* "diagnostic functionality" without *scattering* information all over a complex model.

Finally, the approaches differ technically with respect to the coupling of different "reasons" for adding variability. Let us assume, in the above example, we not only want to add to each sensor a single optional diagnostic functionality (making the sensors' values accessible from an external interface) but also a second function (e.g. logging of the sensors' values for internal error handling purposes). If each of these functions is optional and they can also be combined, we would need to define for each variation point 4 variants: none of the two, 1st functionality, 2nd functionality and the combination of both. In general, whenever n optional and combinable alterations occur at a single point of the model, 2 to the power of n variants need to be specified. This combinatory explosion can be avoided with aspects, because with aspects each of the above functions can be defined separately and they can be combined. So, instead of 4 variants, only two optional aspects would need to be defined: one for the external accessibility of sensor values and one for the internal logging. It is important to mention that this is not true in all circumstances. There are cases where a combinatory explosion can be avoided with the variation point / variant concept (i.e. when it is possible to define one variation point which optionally implements the 1st functionality and a second, separate variation point that optionally implements the 2nd functionality) and, on the other hand, there are cases in which aspects cannot be defined truly orthogonally (discussed below in Section 6.2). But in general, it is true that the definition of aspects is more orthogonal in the sense that it aims at clearly keeping apart different concerns.

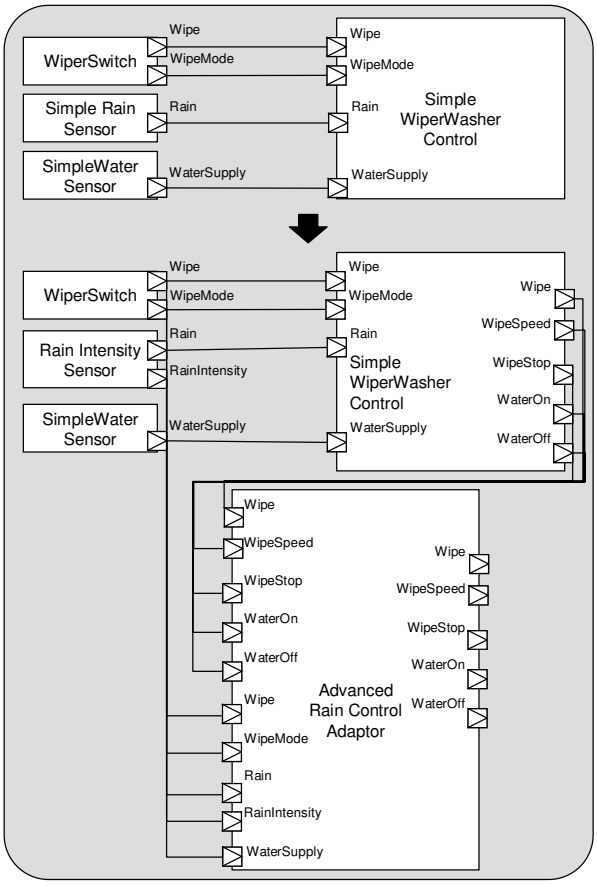

Fig. 6. Wrapper Aspect.

In summary, we have identified these technical differences of the AO approach with respect to the traditional one:

1. unaltered base model
2. no "scattering" of cross-cutting concerns
3. orthogonal definition of optional functionality

6.2. Methodological Implications

After having identified the technical differences of the approaches, we can now turn to a discussion of the implications for automotive software development and the general potential and possible problems of applying aspect-orientation in the automotive context.

Let us start with a relatively simple but nevertheless significant benefit of AO. Since this approach leaves the basic model unchanged, i.e. no explicit variation points need to be specified in the basic model, it can be quite easily realized with legacy tools that do not provide support for variability. This is an important factor for the automotive domain, where a wide range of tools are being applied and shifting to other tools is extremely difficult or impossible for methodological, financial and organizational reasons. Similarly, this property of AO also facilitates the use of legacy models, which is also of great significance in the automotive context.

Furthermore, the clear separation of a base model defining the standard functionality and the separate definition of changes to that basis (through aspects) makes the models – both the base model as well as the definition of alterations – a lot more readable and maintainable. This is especially true for domains where the basic functionality is the most complex part while there exist a very large number of relatively simple variations from that basis. In the automotive domain this is the case, especially for body electronics. Also, the very long evolution of the same models, as is the case in the automotive domain, is very well supported with AO, because when a certain alteration (aspect) is added or is no longer used, this does not alter the definition of the other alterations (aspects).

Another benefit of AO is related to dependencies between variants, which is, as stated above, a significant challenge in automotive development. Since AO allows to aptly modularize cross-cutting concerns, such dependencies will be manageable in a more straightforward manner.

Finally, and most importantly, AO may prove as a very good tool to support an organizational model which is becoming increasingly important in the automotive domain: cross-line development, i.e. development of sub-systems for more than one vehicle line. With AO, the basic model of a sub-system could be developed for several vehicle lines and each vehicle line could introduce its local changes through aspects. The important point here is that the aspects introduced by one vehicle line do not affect those of another.

The same is true not only for separate vehicle lines but also for separate teams working on different aspects (variants) of the same vehicle line. Thanks to the orthogonal definition of aspects they can work relatively independently of one another.

However, some important reservations have to be made at this point. We stated above that aspects can be defined orthogonally, i.e. one aspect (both advice and point-cut) can be defined independently of another and afterwards, both can be combined by weaving both aspects into the same base model. But this is only entirely true for simple cases, such as our diagnostics example from Section 6.1: The internal logging of sensor values is by nature independent of providing these values at some external interface. But what if an optional aspect's functionality has an effect on the base model's behavior? Then other aspects' precise behavior may depend on whether this aspect is actually woven in or not. This is an important concern for further research on applying AO in the automotive domain. In particular, it has to be identified what forms of interdependencies may occur between aspects and how teams working on the aspects separately can be coordinated.

Similarly, all benefits of AO for the automotive domain identified here depend on a clear, readable and maintainable concept for aspect definition. This is, of course, heavily dependant on the type of artifact for which variability is to be defined.

While the use of model-based specification is growing in the automotive domain, there are traditionally still many artifacts, for example requirements or test specifications, which are largely textual and do not have a very sophisticated internal structure. It is an interesting topic of future research to investigate the applicability of the concept of point-cuts and join-points for these artifacts.

7. Related Work

Up to now, the main focus has been on using aspect-oriented techniques on programming language level for implementing variability in product-lines. The approach by Spinzyk et al. starts from feature models and uses an aspect-oriented extension of C++ to implement a product-line [LSS05].

In a similar way, Mezini et al. have examined the appropriateness of aspect-oriented programming languages and of mixin-style programming languages [MO04] for implementing variability of crosscutting concerns. Loughran et al. have proposed a generative approach called framed aspects to instantiate aspects to a specific context for implementing variability and to improve evolution of software product lines [LRZ04], [LR04].

The approach sketched in this paper goes beyond the implementation level by incorporating aspects already as a technique for modeling variability. In addition, it envisions embedding aspect techniques in large-scale product-lines spanning many artifacts of different type. Using aspects this way was also

proposed by Greenfield et al. in [GS05] but without discussing the special characteristics of the automotive domain.

8. Summary and future work

In this paper we have given a brief overview of software product-lines and aspect-orientation and showed where and how aspect-orientation may be utilized within product-line oriented automotive software development. Then, a comparison of traditional and aspect-oriented definition of artifact variability was given, followed by a discussion of the potential of the latter for the automotive domain. While doing so, two issues where identified to be the most important target for future research, namely the precise technique for defining aspects in automotive development artifacts (esp. component diagrams) and the interdependence of aspect definitions together with the possibility of combining several aspects. Since the exemplary aspect specification diagrams in Section 5 de facto describe model transformations, existing techniques and tools for model transformation will prove a useful aid in this research.

Overall, we feel that aspect-orientation may become a valuable instrument to meet some of the challenges specifically encountered in automotive software development. Nevertheless, substantial research in the above mentioned direction still is necessary before this relatively new paradigm can be applied on a broad scale in this domain.

9. References

[BCC05] Berg, K.v.d., Conejero, J.M., Chitchyan, R.: "AOSD Ontology 1.0 – Public Ontology of Aspect-Orientation". Technical Report AOSD-Europe Deliverable D9, AOSDEurope-UT-01, Universiteit Twente, 2005.

[GS05] Greenfield, J., Short, K.: "Software Factories". Wiley, 2005.

[Her02] Herrmann, S.: "Object Teams: Improving Modularity for Crosscutting Collaborations". Proc. of Net.ObjectDays, Erfurt, 2002.

[LRZ04] Loughran, N., Rashid, A., Zhang, W., Jarzabek, S.: "Supporting Product Line Evolution with Framed Aspects". 3rd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS 2004), AOSD'04, 2004.

[LR04] Loughran, N., Rashid, A.: "Framed Aspects: Supporting Variability and Configurability for AOP", International Conference on Software Reuse (ICSR-8), Madrid, Spain, 2004.

[LSS05] Lohmann, D., Spinczyk, O., Schröder-Preikschat, W.: "On the Configuration of Non-Functional Properties in Operating System Product Lines". Proceedings of the 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS 2005), Chicago, IL, USA.

[Mil04] Miles, R.: "AspectJ Cookbook", O'Reilly 2004.

[MO04] Mezini, M, Ostermann, K: "Variability Management with Feature-Oriented Programming and Aspects", in: Foundations of Software Engineering (FSE-12), ACM SIGSOFT, 2004.

[PBL05] Pohl, K., Böckle, G., van der Linden, F.: "Software Product-Line Engineering". Springer, Heidelberg, 2005.