From Aspect-Oriented Models to Aspect-Oriented Code? The Maintenance Perspective

Aram Hovsepyan, Riccardo Scandariato, Stefan Van Baelen,
Yolande Berbers, Wouter Joosen
DistriNet, Katholieke Universiteit Leuven
Celestijnenlaan 200A
BE-3001 Leuven, Belgium
{first.last}@cs.kuleuven.be

ABSTRACT

Aspect-Oriented Modeling (AOM) provides support for separating concerns at the design level. Even though most AOM approaches provide means to execute the composition of the modularized concerns to obtain a composed model, it is also possible to keep the concerns modularized at the implementation level by targeting an aspect-oriented platform. Model-driven approaches have emerged to support both alternatives via tools. Clearly, these choices are not equivalent. Rather, they have a direct impact on several dimensions, including maintainability. Hence, the main research problem addressed by this work is to figure out which alternative provides for shorter maintenance time. To answer this question, we have conducted a series of quantitative studies and experiments.

Categories and Subject Descriptors

D.2.10 [**Software Engineering**]: Design—*Methodologies*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Ehnancement, Extensibility*; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Experimentation, Languages

Keywords

Aspect-oriented modeling, model-driven engineering, model composition, aspect-oriented programming, empirical study

1. INTRODUCTION

Aspect-oriented software development (AOSD) is an emerging set of methods and technologies that allow multiple concerns to be modularized, expressed, composed and analyzed in isolation. So far, the most prominent applications of the aspect-oriented (AO) paradigm emerged from the implementation phase of the software life cycle. Aspect-Oriented Programming (AOP) is a programming paradigm that increases modularity by allowing the separation of crosscutting code-level concerns as first-class elements, called aspects [18]. Recently, AO ideas are making their way through earlier stages of the software development process, such as requirements engineering [24, 33], architectural modeling [28] and design [3, 25, 35]. Zooming into design-level approaches, Aspect-Oriented Modeling (AOM) provides support for separating concerns at the design level and has the potential to effectively tackle the complexity of developing software that deals with interdependent concerns [16].

The AOM research community is trying to deliver techniques that reduce (if not eliminate) the need for lower level AO artifacts like pointcut specifications. For instance, most AOM approaches provide means to specify and execute the composition of the modularized concerns at the modeling level to obtain a combined model. However, it is also possible to keep the concerns modularized at the implementation level by targeting an AO platform. This paper studies and analyzes the value of (or the need for) adding AO code-level artifacts in light of the maintenance perspective.

Tools have been developed that leverage AO models to generate both AO and object-oriented (OO) implementation code [11]. Therefore, a full spectrum of options is now available to developers: they can pick from the all-aspectual process where AO models are translated to AO code, the all-object-oriented process, and two types of hybrid processes (AO models with OO code, and vice versa). Clearly, these choices are not neutral. Rather, they have a direct impact on several dimensions, such as productivity of developers, quality of the final product, ease of maintenance of the artifacts, and several other matters.

This paper focuses on aspect-oriented modeling and investigates the effect on maintainability of two process alternatives: an all-aspectual process where Theme (AO) models [3] are translated into AspectJ [18] (AO) code is compared to a hybrid process where Theme (AO) models are translated into Java (OO) code. For instance, the aspectual process would be the obvious choice for a development organization that has fully embraced the AO paradigm. The hybrid process is more conservative and preserves the know-how that the development team might have built up over several years of object-oriented coding. To summarize, the main research problem is to figure out which process alternative allows shorter maintenance time.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AOSD'10 March 15–19, Rennes and St. Malo, France
Copyright 2010 ACM 978-1-60558-958-9/10/03 ...$10.00.

In a Model-Driven Engineering (MDE) approach, a maintenance task involves the modification of both the models and the generated reference code (eventually with iterations). Hence, the intuition suggests that the aspectual process delivers more with respect to maintenance, as there is no asymmetry between the paradigms used at both design level and code level. For instance, the correspondence between the model abstractions (e.g., themes) and the code abstractions (e.g., aspects) simplifies the task of locating the place in the code base that is affected by a change previously injected at the model level (e.g., to finalize the maintenance task with code edits). Note that we do not consider AOM approaches that support full code generation (e.g., [10, 37]), as these are not considered to be representative for AOM [16]. We also do not take into account MDE approaches that allow immediate code modifications by automatically reflecting those changes at the model level (e.g., [26, 41]), as the support for round trip engineering in AOM is currently very limited.

In order to build a solid argument supporting the intuition above that the all-aspectual process has a number of advantages as far as maintenance is concerned, we conducted a study on two applications, namely an on-board vehicle system that automates the payment of tolls for vehicles traveling on highways, and a heart pacemaker system. The study is articulated into two parts. First, we applied the two alternative processes to both applications and we gauged the resulting artifacts along several internal attributes (size, coupling, cohesion and separation of concerns) that may be predictors of maintenance. The collected metrics confirmed the initial insight of the aspectual process outperforming the hybrid process. In the second part of the investigation, we conducted a series of controlled experiments where 10 subjects executed actual maintenance tasks on both applications using the two alternative processes. This study does not analyze all facets of maintainability. The focus is on perfective maintenance tasks that add new functionality or improve on existing functionality. Furthermore, the size of the maintenance tasks is limited (although adequate with respect to the size of the applications) and the tasks require relatively local modifications. The results show that, on average, the aspectual process allows shorter maintenance times, although the competitive edge is not vast.

The rest of this paper is structured as follows. In section 2, we sketch the problem statement in detail. In section 3, we present the objectives of the study and outline the evaluation methodology. We present a high-level overview of the applications in section 4. In section 5, we discuss the first part of the study where internal quality attributes are assessed. In section 6, we present the user experiments that we have conducted along with a statistical analysis of the obtained results. Sections 7 and 8 conclude this paper by presenting the related work and summarizing our findings.

2. PROBLEM STATEMENT

A typical AOM approach allows one to model the system's concerns separately and specify their composition using a composition model (see step 1 in figures 1 and 2). Given such an AOM approach (in our case, Theme), there are in general two alternative processes to produce a working system starting from an aspect-oriented (modularized) design.

2.1 Aspectual Process

The first alternative (depicted in fig. 1) is to keep the concerns modularized all the way down to the source code. A model-to-code transformation tool (e.g., [32, 39]) is used to transform each modularized concern to the code level (in our case, AspectJ). The composition model is reflected at the code level by a set of pointcuts. The composition itself is performed by the AO source code weaver that produces a composed byte-level code (not shown on the figure). Several AOM approaches follow this alternative (e.g., see [3, 20, 27]). We refer to this approach as the *aspectual process*.

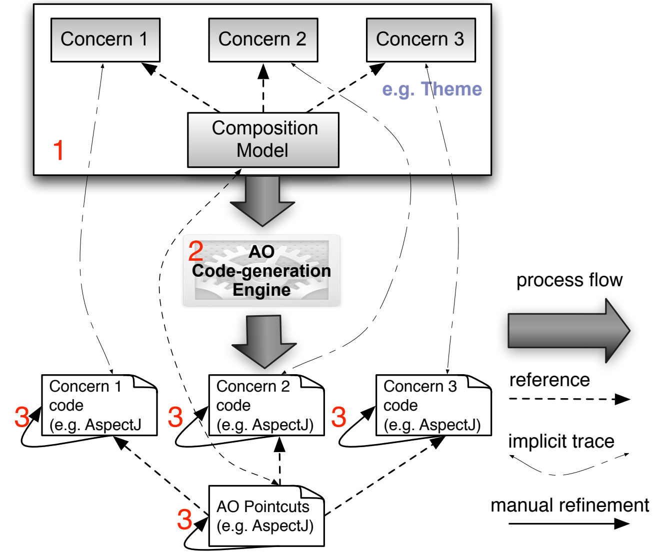


Figure 1: Aspectual process

In summary, the aspectual process requires the execution of the following steps (see the numbers in fig. 1):

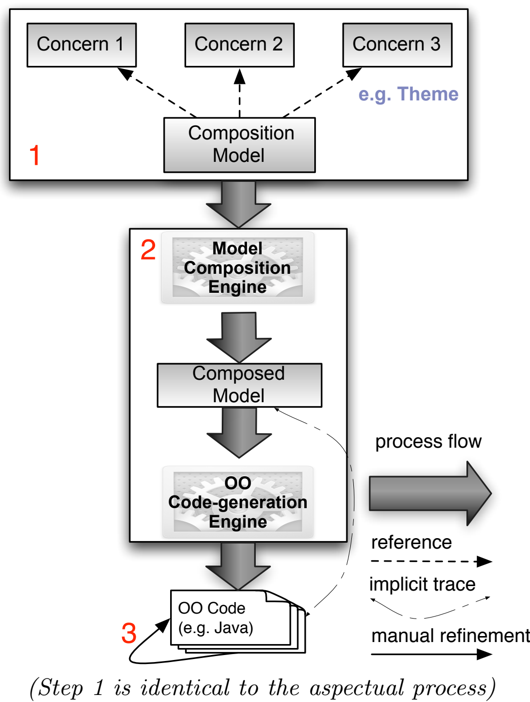
1. Design the separate concerns and specify their composition
2. Generate the reference code by running the automatic model-to-code transformation tool
3. Refine the generated code manually in order to obtain a complete system.

The first step requires one to create the concern models and specify how the concerns should be composed. The composition specification depends on the given AOM approach and could be an explicit composition model (as illustrated on the figure), a list of heuristics implied by the methodology (e.g., name-based or pattern-based merging), a textual composition description, etc. The methodology to create the design model is out of the scope of this work (some AOM approaches provide such a methodology [3]). Once the design is in place, one can transform the modeling constructs to the code level by using a model-to-code transformation tool (step 2). Finally, the generated code should be manually refined in order to produce a complete system (step 3). These refinements typically consist of completing the behavior of automatically generated methods or advices. In the context of this study, the code may not be modified in such a manner that it is incompatible with the model (e.g.,

by adding a new class in the code). One must always go back to step 1 in case such modifications are necessary. The same applies to the maintenance cycles, i.e., all maintenance tasks must always start from step 1 rather than executing modifications immediately at the code level.

2.2 Hybrid Process

The second process alternative (depicted in fig. 2) starts from the same aspectual model as before. This time, however, the composition of the modularized concerns is performed at the modeling level (see top part of box 2). This composed model is then fed to a model-to-code transformation engine that produces OO source code (in our case, Java). Even though model composition does not necessarily imply the use of OO programming languages, it is the closest match in terms of abstractions used. A number of existing AOM approaches imply that this option is followed (e.g., see [4, 23, 25, 35]). Throughout the paper we refer to this approach as the *hybrid process.*



(Step 1 is identical to the aspectual process)

Figure 2: Hybrid process

The hybrid process requires the execution of the following steps:

1. Design the separate concerns and specify their composition
2. Compose the modularized concerns by using a model-to-model transformation tool and generate reference OO code by running a model-to-code transformation tool
3. Refine the generated code manually in order to obtain a complete system.

Step 1 in the hybrid process is identical to step 1 in the aspectual process. Once the design is in place, model composition is performed and followed by code generation (step 2). Finally, step 3 requires the completion of the partially generated OO code. Identical to the aspectual process, all maintenance operations must be always performed starting from the model, i.e., step 1. Note that the composed model in step 2 is mainly used for understanding and analyzing the interactions across the composed concern models [16]. Even though in theory it is possible to have a process that allows one to refine the composed model before generating code, this is out of scope for this work.

As far as maintenance is concerned, there are two main differences between the two processes. Firstly, the aspectual process preserves the AO paradigm throughout all layers of the process, while the hybrid process mixes AO modeling concepts with OO coding. As a result, developers who use the hybrid process have to make the mental shift between a particular way of describing the design (AO modeling) and a different way of implementing the software (OO coding). Hence, we expect that maintenance will require more effort from the developers in the hybrid case. Secondly, from a pure coding perspective, a number of recent quantitative studies have suggested that in certain cases AOP maintenance cycles may require less effort compared to OOP [1, 19, 44]. Hence, we expect that the aspectual process will provide better code in terms of internal quality attributes. Better code quality is often considered to be a predictor of smaller maintenance cycles (although a strong proof is yet to come).

Thus, the research problem we are investigating is whether the aspectual process, which maintains the AO paradigm throughout the different development steps, offers benefits in the context of system maintenance compared to the hybrid process, which makes the leap from AO models to OO code.

2.3 Technology Selection

This section elaborates on the motivations behind the selection of Theme, AspectJ and Java as the pillars of the aspectual and hybrid processes. The choice of Theme (which is used in both processes) was motivated by a number of factors. Firstly, Theme is one of the most mature AOM approaches available in the literature. Secondly, Theme comes with abundance of documentation, which is essential to make the experiments repeatable. Moreover, Theme features relatively strong tool support that is open to modification. Finally, many of the experiment subjects were to some degree familiar with Theme. The selection of AspectJ for the aspectual process was also influenced by its maturity and availability of good documentation. In addition, all experiment subjects were familiar with AspectJ. Finally, Theme, AspectJ and Java were chosen because they are regarded as representative in the community for AOM, AOP and OOP techniques respectively.

3. GOAL AND GENERAL APPROACH

The overall aim of this evaluation is defined as follows:

Goal. Investigate whether the use of the aspectual process based on the Theme approach leads to shorter or longer cycles for maintenance tasks than the hybrid process.

We only consider maintenance tasks that add new functionality or improve on existing functionality, and that involve relatively local changes.

To this aim, we have devised two tracks of quantitative investigation:

1. *Measuring the predictors of maintainability.* For two reference applications (introduced in section 4), we applied the two alternative processes starting from the Theme models of the applications. The resulting implementations have been compared by measuring some internal quality attributes. As maintainability is assumed to be predicted by size (the smaller the better), coupling (the lower the better), cohesion (the higher the better), scattering (the less scattered the better) and tangling (the less tangled the better), we have selected a metrics suite that allows us to investigate the following three sub-hypotheses:

   (a) the implementation obtained by using the aspectual process is smaller in terms of size compared to the hybrid process

   (b) the implementation obtained by using the aspectual process is less complex in terms of coupling and cohesion compared to the hybrid process

   (c) the implementation obtained by using the aspectual process is less scattered and less tangled in terms of separation of concerns compared to the hybrid process

2. *Measuring of maintenance effort.* Via an experimental study we want to investigate the hypothesis that, on average, the overall time necessary to understand and implement a change to the system is shorter using the aspectual process compared to the hybrid one.

The first track will help us to gain an initial insight whether the aspectual process is more beneficial than the hybrid process. However, the data obtained from the two applications will not be sufficient to have a statistical significance. Moreover, the effect of the internal quality attributes on maintainability of a system is only hypothesized in the literature. Hence, the second track will provide a more experimental proof of the benefits of the aspectual process.

Table 1: Size, coupling and cohesion, and separation of concerns metrics for the internal quality attributes

	Metrics	Definition
Size	Lines of Code (LOC)	Counts the total number of lines of code not including the comments and white lines.
	Number of Attributes (NOA)	Counts the number of attributes of each class or aspect.
	Weighted Operations per Component (WOC)	Counts the number of methods and advice of each class or aspect and the number of its parameters.
Coupling	Coupling Between Components (CBC)	Counts the number of other classes and aspects to which a class or an aspect is coupled.
Cohesion	Lack of Cohesion in Operations (LCOO)	Counts the number of pairs of operations working on different class fields minus pairs of operations working on common fields (zero if negative). It is equal to zero when all methods don't access any field.
SoC	Concern Diffusion over Components (CDC)	Counts the number of Java or AspectJ classes/aspects whose purpose is to contribute to the implementation of a concern and the number of classes/aspects that access them.
	Concern Diffusion over Operations (CDO)	Counts the number of methods and advice whose main purpose is to contribute to the implementation of a concern and the number of other methods and advice that access them.
	Concern Diffusion over LOC (CDLOC)	Counts the number of transition points for each concern through the lines of code. Transition points are points in the code where there is a "concern switch".

3.1 Predicting Maintainability

In order to measure the internal quality attributes we use a number of known metrics available in the literature.

We use a set of traditional coupling, cohesion and size metrics [6] to measure the implementation size, coupling and cohesion. As these metrics have been originally designed to measure the internal quality attributes for OO code, for the aspectual process we use an extension of these metrics that support AO concepts [38]. In order to measure the implementation scattering and tangling, we use the Separation of Concerns (SoC) metrics [38]. These metrics are used and proven to be effective in a number of systematic quantitative studies that analyze how AOP promotes superior separation of concerns in the implementation of crosscutting features [1, 5, 7, 14, 15, 19, 40]. Table 1 presents an overview of the metrics we used and provides the definition of each one. These metrics have been collected automatically using an adjusted version of the aopmetrics tool [2].

3.2 Assessing Maintenance Effort

In order to measure which of the two processes provided better support over maintenance cycles, we have mapped (according to the literature [44]) the notion of maintainability into an effort metric.

The time necessary to execute the maintenance tasks is used to estimate the maintenance effort. We refer to this measure as the EFFORT in the rest of the paper. This metric is measured in a series of controlled experiments executed by a number of experimental subjects. As detailed later, each subject has been supervised during the execution of the experiment. In order to measure the EFFORT, we have developed a web-based time recorder tool that was used by the experiment supervisor to start/pause/stop the time tracking per maintenance task.

Note that the EFFORT measure includes both the understanding time and the change execution time. Indeed, the most common approach when performing a maintenance task is to switch back and forth between understanding and "changing", thus it is quite difficult to identify (and hence, measure) the two activities apart.

The detailed description of the experimental setup, the profiles of the participants, and the maintenance tasks are given in Section 6.

4. APPLICATIONS

In both tracks mentioned above, two applications (toll payment system and pacemaker system) are used as objects of experimentation. The applications are taken from two projects created by the Theme team [11][1]. These applications have been selected because they are relatively small and can be handled by the subjects in the context of a supervised user experiment. Despite their size, these are realistic applications from different domains that are known to challenge SoC in the design (e.g., see the real-time concern in the Toll System; and the security, synchronization and fault tolerance concerns in the Pacemaker application). As any realistic software system from the majority of domains is likely to have concerns that cannot be modularized along a single dimension [34, 43], the above selection represents a good playground for the purposes of our study.

4.1 Application 1: Toll System

The Toll System application is based on a real product line for the toll payment automation of vehicles traveling on toll roads [13]. The system consists of back office toll servers that are responsible for charge data collection. Each vehicle is equipped with an on-board unit that includes a GPS. The on-board unit automatically detects toll roads, collects the journey data and sends it back to the toll servers using a GSM/GPRS network at the end of the journey. The Toll System is modularized into two base themes and one crosscutting aspect theme (see table 2).

Table 2: Themes in Toll System design

	Themes	Main responsibilities
Base Themes	Charge Calculation	Calculate the toll charge for a given vehicle based on the list of positions that define a journey.
	Road Recognition	Using a built-in GPS receiver determine the road the vehicle is driving on.
Aspect Themes	Real-Time Data Display	Display the acquired road information and the calculated charge data on the on-board screen.

Figure 3 illustrates the composition model that provides further insight into the application structure. The composition model specifies that base themes *ChargeCalculation* and *RoadRecognition* should be composed together, while the crosscutting *RealTimeDataDisplay* should be bound with the two base themes.

4.2 Application 2: Pacemaker

A pacemaker is an embedded medical device designed to monitor and regulate the beating of the heart when it is

[1]The complete Theme design models, sources, transformation and code generation engines, participant profiles and other relevant data used in this paper can be found at http://www.cs.kuleuven.be/~aram/process_study.html

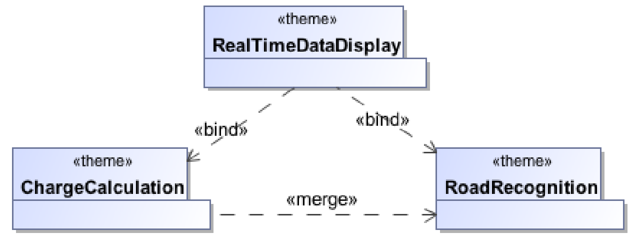


Figure 3: Composition model for the Toll System

not beating at a normal rate. The pacemaker under consideration has typical requirements, such as sensing the heart rate, analyzing the heart rate and generating a pulse. It is also required to wirelessly communicate diagnostic information and to support device update. The system consists of five base themes encapsulating the core behavior, and three aspect themes that represent non-functional requirements cutting across the base themes. Table 3 shows each theme and outlines the responsibilities.

Table 3: Themes in Pacemaker design

	Themes	Main responsibilities
Base Themes	Monitor Heart	Provide access to heartbeat and heart temperature sensor data.
	Analyze Pulse	Analyses the frequency of the heartbeat to determine the current heart rate.
	Generate Pulse	Adjust the heart rate based on knowledge of the ideal rate and the current rate.
	Acquire Data Externally	Provide remote, wireless access to the pacemaker and access to diagnostic information.
	Update Pacemaker	Remotely update the pacemaker firmware.
Aspect Themes	Security	Provide data security on outgoing communications via data encryption.
	Synchronization	Support critical sections via mutual exclusion.
	Fault Tolerance	Handle sensor errors by coordinating between multiple implementations of the same sensor access behavior.

Figure 4 provides a structural overview of the Pacemaker application by illustrating the composition model, which specifies that all base themes should be merged together, while the aspect themes should be bound to base themes as specified by the requirements.

5. PREDICTING MAINTAINABILITY

This section reflects on the results of the internal metrics we have obtained from the implementations of the two applications. The applications have been implemented by the authors using both processes presented in section 2. Before collecting the metrics, we have manually aligned the AspectJ and Java implementations in order to ensure that coding styles and implemented functionality are the same.

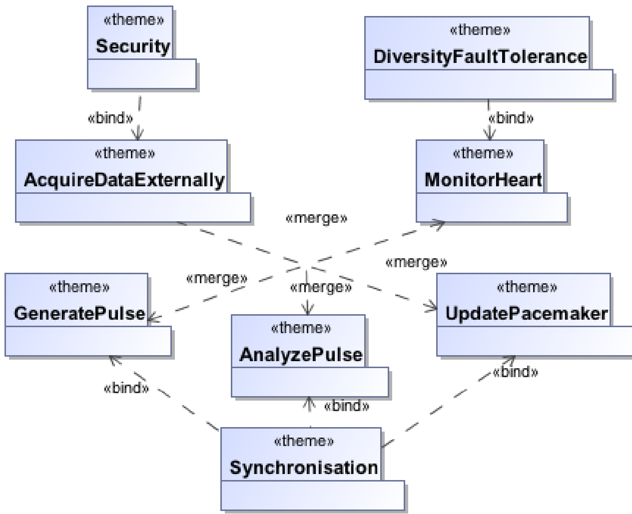


Figure 4: Composition model for the Pacemaker

5.1 Results

Table 4 presents the collected absolute values for SoC metrics. All the numbers favor the aspectual process that keeps the concerns separated all the way to the AspectJ source code. In summary, our results align with previous studies (e.g., [1, 19]) that have demonstrated that AO code performs generally better with respect to the considered SoC metrics. In more details, however, the differences between the two approaches are smaller for the Toll System compared to the Pacemaker. This is possibly explained by the higher number of crosscutting concerns in the Pacemaker application.

Table 4: Absolute values for Separation of Concerns metrics

	CDC	CDO	CDLOC
Toll System (hybrid process)	23	92	54
Toll System (aspectual process)	22	82	44
Pacemaker (hybrid process)	25	93	54
Pacemaker (aspectual process)	17	66	23

Table 5 shows the results for the traditional OO metrics used to measure the size, the coupling and the cohesion of a system. As opposed to the SoC metrics, none of the existing quantitative studies in the literature (e.g., [1, 19, 44]) concludes that AO code always outperforms OO code when it comes to the traditional metrics. Typically, AO code performs better only in specific cases, while in some cases there is no difference or AO is actually inferior to OO. The reason is that AO code shows increased coupling due to the phenomenon known as the "fragile pointcut" problem [42]. In our study, however, almost all the numbers favor the aspectual process. A possible explanation for these results is that the model-to-code transformation engine systematically uses a limited subset of AOP constructs in the generated code. We also believe that the misalignment between Theme, which is a symmetric AO approach [22], and AspectJ, which is asymmetric [22], has slightly affected the results of the above metrics. In order to overcome the above-mentioned misalignment, the model-to-code transformation tool generates duplicated empty class files for each class or aspect implementation. Hence, this negatively influences both LOC and CDC metrics for the aspectual approach.

Table 5: Absolute values for coupling, cohesion and size metrics

	LOC	NOA	WOC	CBC	LCOO
Toll System (hybrid process)	363	34	72	14	294
Toll System (aspectual process)	372	33	68	12	226
Pacemaker (hybrid process)	369	29	77	9	276
Pacemaker (aspectual process)	362	27	68	7	125

In summary, the results so far provide an indication that maintenance cycles using the aspectual process will be shorter than using the hybrid process. In the next section, we discuss an experimental study that we have conducted in order to obtain a more statistical proof of these initial indications.

6. EXPERIMENTAL STUDY

We first describe the maintenance tasks that the subjects had to perform on each application. Then we describe the experimental design and present the results along with the statistical analysis.

6.1 Maintenance Tasks

The experimental study has been conducted on two applications introduced earlier. We have devised two maintenance tasks per application. For the selection of the tasks, the rationale is to have tasks of comparable (although not equal) difficulty across the applications and with a balanced mix of changes that affect both modularized and crosscutting concerns. We have only considered two main types of maintenance tasks, i.e., functionality addition and functionality improvement. Table 6 presents the tasks in details and summarizes the type of impact of the changes.

6.2 Experimental Design

We have designed an empirical study in order to measure the maintenance effort required to complete a set of maintenance tasks, executed using both processes. We enrolled 10 volunteering participants. The 10 subjects participated one by one in ten supervised experimental sessions. For the purpose of analysis, the subjects have been matched into 5 pairs, although each participant executed the tasks in isolation. In fact, the participants are not even aware of the identity or results of their pair-mate. The pairing approach was used for statistical purposes only, as it allows to reach more powerful results with a smaller data set.

Subjects of each pair have comparable skills, experience, and background knowledge. All ten subjects in the experi-

Table 6: Summary of the maintenance tasks

	Task number and description	Impact
Application 1 (Toll System)	T1. Introduce different fare calculation strategies that take into account additional parameters (ecological classification, vehicle properties, etc.).	Involves a change that is local to a base concern and it is reasonably easy to identify which theme is involved
	T2. Upload the information on vehicle position and current charge to the data center.	Requires the addition of a new aspect theme and, hence, some changes to the composition are in order
Application 2 (Pacemaker)	T3. Analyze the pulse using a more complex heart rate model depending on a number of additional parameters (patient age, patient life style, etc.).	Demands for a change that is local to a base (i.e., non crosscutting) theme, but it is not obvious which one
	T4. Log the pulse and temperature values for all heart monitors whenever a new value is sensed.	Necessitates an existing aspect theme to be modified

ment were post-graduate researchers at our lab, where seven of them had at least 4 years of *seniority*. All of them had previous *experience with Java*. In addition, all subjects were *familiar with AspectJ*, but none of them had actual coding experience with this technique. All participants were familiar with the UML, while three of them were also *familiar with Theme*. Table 7 presents the exact pair compositions. Note that, given the amount of code writing required, the Java skills were preferential over the subject seniority.

Table 7: Composition of the experimental subjects

Pairs	Java experience	Post-graduate seniority (in years)	AspectJ skills	Theme skills
1	very good	4, 1	little	none
2	excellent	1, 4	little	none
3	excellent	4, 3	little	none
4	very good	4, 5	little	none, little
5	very good	4, 4	little	little

The experiment has been designed so that each subject performs two tasks from one application and two tasks from the other application, but using opposite processes with respect to his pair-mate. Within each pair the tasks were executed exactly in the same order. However, in order to avoid any bias due to the order in which the pairs execute the tasks and the order of the used processes, we have opted for a balanced experiment setup (i.e., the first three pairs started from Application 1 and the last two pairs started from Application 2). This experiment setup is similar to the one described by Tonella and Ceccato [44] and limits the influence due to the order of treatments. Table 8 summarizes the experimental design.

Table 8: Experiment design

Pair	Subject	Tasks order
1	1	T1, T2 (aspectual); T3, T4 (hybrid)
	2	T1, T2 (hybrid); T3, T4 (aspectual)
2	3	T1, T2 (aspectual); T3, T4 (hybrid)
	4	T1, T2 (hybrid); T3, T4 (aspectual)
3	5	T1, T2 (aspectual); T3, T4 (hybrid)
	6	T1, T2 (hybrid); T3, T4 (aspectual)
4	7	T3, T4 (aspectual); T1, T2 (hybrid)
	8	T3, T4 (hybrid); T1, T2 (aspectual)
5	9	T3, T4 (aspectual); T1, T2 (hybrid)
	10	T3, T4 (hybrid); T1, T2 (aspectual)

Note that the given experimental setup mitigates any potential learning effects as far as the applications and the methodologies are concerned. Every participant uses both processes, however in different applications. The same task is never executed by the same participant twice, with any process. In addition, the modeling effort required in both processes is exactly the same. Hence, both subjects in a pair improve their modeling skills equally, as they progress in the experiment.

6.3 Interview

At the end of the experimental sessions, each subject has been shortly interviewed with the aim to obtain the subjects' perspective on several parameters of the study. Table 9 presents the questions asked during the interview.

Table 9: Interview questions

	Question
Q1	Were all the tasks equally difficult?
Q2	Do you think the aspectual process was easier to handle than the hybrid one or vice versa? And if so, why?
Q3	Did you leverage on the composed model in the hybrid process? If yes, did you find it useful?
Q4	Additional remarks

Q1 is meant to validate the experiment setup concerning the comparable difficulty and size of the tasks. The objective of Q2 is to obtain the subjects' perception on whether one of the two approaches is easier to use, as to (possibly) corroborate the results of the EFFORT measure. Q3 investigates whether participants' experience supports the hypothesis that an explicit composed model is one of the main expected advantages of the hybrid approach. Finally, in the last part of the interview the subjects are free to provide any additional remarks (Q4).

6.4 Experiment Execution

The maintenance tasks were executed on a workstation using MagicDraw [30] for modeling and Eclipse for model transformations, code generation and coding. During a try-out phase (prior to the actual experiment execution) the complete toolset was modified and fine-tuned by the authors to ensure that the produced AspectJ and Java code were aligned and equivalent. The effort required from the participants in order to use the toolset in both processes was roughly the same (same amount of "button clicks"). More-

over, the supervisor provided sufficient assistance to minimize the overhead generated by the toolset usage. Our assumptions were confirmed during the interview mentioned in the previous section. Only two subjects have suggested in their additional remarks that the usability of the toolset could be improved. However, there were no remarks as far as the produced code is concerned. In addition, the usability difficulties were common to the tool chains of both processes.

Before the experiment started, each subject was given a short tutorial on AspectJ and Theme. In addition, each participant was asked to implement a toy application using both approaches in order to obtain more hands-on experience with both approaches and the toolset.

At the beginning of the experiment, the participant was given a short introduction to the first application. The functionality was explained by the supervisor via slides and the full documentation was provided, i.e., both the Theme model and the implementation code. Then the participant was provided with a textual document describing the first assigned maintenance task. The participant was asked to implement each change by first adjusting the model and then refining the automatically generated code. The participant was allowed to go back to the model and iterate over the process, although such behavior has never been observed in the experiments, possibly due to the moderate size of the applications. The supervisor started the timer at the point when the task was handed out and stopped it when participant indicated that he was done with the task. Then the correctness of the produced artifacts was ensured by means of manual inspections performed by the supervisor. In case any errors were found, the errors were mentioned to the subject after the inspection and the timer was set forth until the successful completion of the task. The above setup was repeated for the second task. Afterwards, the supervisor described the second application and the experiment was repeated with two additional tasks. For these tasks, the participant had to switch to the other process.

6.5 Statistical Method

We want to study the correlation between the EFFORT (time) and the process used (aspectual or hybrid) per task. Specifically, we are interested in knowing whether in each task there is a statistically significant difference among the average EFFORT of the two groups (each using a different process) that executed the task. Note that we never aggregate the samples from different tasks to avoid bias. As it is not possible to assume a normal distribution for the collected measures, we resorted to the non-parametric tests. The appropriate statistical test for our setup (which involved paired samples) is the Wilcoxon signed-rank test. In summary, the null hypotheses are H0: $\mu^{Ti}_{\mathrm{ASPECTUAL}} - \mu^{Ti}_{\mathrm{HYBRID}} = 0$ for i=1..4. The output of the correlation analysis is a p-value, and we reject the null hypothesis for $p < 0.05$.

6.6 Results

Tables 10 and 11 present the measured EFFORT in seconds, spent per application.

The results show that the aspectual process performs consistently better over all the tasks. We observed a 6.1% reduction of EFFORT in Task 1, a 19.3% reduction in Task 2, a 16.1% reduction in Task 3, and a 9.3% reduction in Task 4. Table 12 summarizes the results of the statistical analysis. According to these p-values, we can assert that the above-mentioned conclusions are statistically significant for Task 2 (Application 1) and close to significant for Task 3 (Application 2).

Table 10: Application 1 (Toll System) – Measured EFFORT in seconds

		EFFORT	
	Pairs	**Aspectual process**	**Hybrid process**
Task 1	pair 1	1986	1864
	pair 2	1429	2187
	pair 3	1840	1788
	pair 4	1902	2038
	pair 5	1786	1645
	Average	1788.6	1904.4
Task 2	pair 1	1008	1320
	pair 2	581	900
	pair 3	781	874
	pair 4	910	1139
	pair 5	885	925
	Average	833	1031.6

Table 11: Application 2 (Pacemaker) – Measured EFFORT in seconds

		EFFORT	
	Pairs	**Aspectual process**	**Hybrid process**
Task 3	pair 1	1296	1579
	pair 2	738	965
	pair 3	1047	1353
	pair 4	1311	1560
	pair 5	1091	1081
	Average	1096.6	1307.6
Task 4	pair 1	683	722
	pair 2	699	536
	pair 3	660	838
	pair 4	763	999
	pair 5	732	803
	Average	707.4	779.6

Table 12: p-values from Wilcoxon signed ranks test

	Task 1	**Task 2**	**Task 3**	**Task 4**
Asymp. Sig. (2-tailed)	0.893	0.043	0.080	0.225

6.7 Answers to the Interview

The objective of Q1 was to look into the perceived difficulty of the tasks. 6 out of 10 participants have mentioned that the tasks were comparable, 2 have noted that both T1 and T3 were slightly more difficult, and one subject mentioned that only T1 was a bit more difficult. This confirms that, overall, the tasks are of comparable complexity. However, T1 and T3 were the first tasks in each application and, therefore, these tasks required the subjects to familiarize themselves with the applications in question. This could explain the perception of the increased difficulty.

Q2 looked into the subjects' perception of whether the aspectual process was better than the hybrid one. 5 out of 10 subjects have mentioned that the aspectual process was better than the hybrid process, 3 noted that both ap-

proaches were comparable and 2 were in favor of the hybrid approach. However, both subjects in favor of the hybrid approach believed their strong background in OO motivates this choice. Overall, these answers align well with the empirical results: the aspectual process has a larger favor although the opinions are not very polarized. Note that, as mentioned, the transformation from Theme design models to AspectJ is slightly complicated by the misalignment between Theme (symmetric) and AspectJ (asymmetric). Most of the subjects mentioned that this was slightly confusing and if a symmetric AOP language was used the aspectual process would be more obvious.

The objective of Q3 was to determine if the presence of an explicit composed view would favor the hybrid approach. However, only 3 out of 10 subjects have effectively used the composed view and found it useful. About the rest of the group, one subject has reported that the composed view might be useful in case of larger models, while two subjects have noted that the composed view may even be counterproductive. We believe that, given the relatively small model sizes, the composition is easy to grasp from the elementary pieces.

Finally, concerning Q4, there were no additional remarks except for the the tool chain usability issues that we have already discussed in section 6.4.

6.8 Discussion

Although the average EFFORT metric favors the aspectual approach, the difference is only statistically significant for task 2 (and task 3 to some extent). This section provides some possible explanations of these results. These insights could be used as working hypotheses for further studies that could replicate or refine this work.

We suspect that the nature of the tasks had an impact on the results. The second task (T2) required the introduction of a new crosscutting theme and adjustments of the composition model. Code-level modifications were required to implement the behavior of this crosscutting advice. In the AO code, this behavior was implemented as a single code-level advice that was injected at two join points in the base code. In the OO version, replication was necessary to implement the same behavior at the two places explicitly. Hence, it is obvious why the aspectual process has outperformed the hybrid one and this was reaffirmed by the experimental results.

The first (T1) and the third (T3) tasks are similar: they both require the modification of an existing base theme. In case of the first task, the code-level modifications were very similar for both processes and required the understanding of only two base themes. Hence, in theory, the expected required effort for both processes should have been roughly the same. This is actually confirmed by the results (no statistically significant difference). In case of the third task, however, the modification to both the model and (particularly) the code required the understanding of three base themes and one aspect theme. We believe that this factor has contributed to getting the numbers in favor of the aspectual process for task 3 with an almost significant statistical difference.

Finally, the fourth task (T4) required a modification (at the design level) to an existing aspect theme where extra functionality was added to the advice. In the OO code, this required scattered modifications and, hence, the aspectual process could have an advantage. However, the extra functionality also required a more permissive pointcut specification to be defined in the model. The generated AspectJ pointcut specification, though, was *too* permissive and some extra coding was necessary in the advice to overcome this problem. As a consequence, the aspectual process lost the previously mentioned advantage. Note that the problem does not lie neither in the generator nor in the semantics provided by the modeling methodology. The task could have been implemented by defining an additional theme and, in such a case, the aspectual approach would have been superior. However, the subjects were explicitly asked to modify the theme that was already existing. In these conditions, defining a too permissive advice is the only way out.

6.9 Threats to Validity

We discern between two types of threats to validity: internal validity threats and external validity threats.

The *internal validity* refers to the extent to which one can accurately state that the independent variable (the different processes) produced the observed effect.

- One of the possible threats to the internal validity is the individual user familiarity with Java, AspectJ, Theme and other tools used in the experiments. This is why we have introduced the subject pairs where each pair is assumed to have approximately the same knowledge and expertise with the concepts used. However, despite our best efforts pairing remains subjective.
- During the execution of the experiment, the subjects may react differently over time. E.g., subjects may become bored or tired, or they may become more or less positive to one or another treatment. However, we did not notice any serious effects and this threat can be discarded.
- The subjects' knowledge of the purpose of the evaluation is also a threat to the internal validity. Hence, participants may not be impartial to the outcome of the experiment. Nevertheless, given the constant supervision, we believe that this threat was minimized.
- Finally, the misalignment between Theme and AspectJ may have impacted the results of the study as well. Still, we believe that this impact is actually in the disadvantage of the aspectual approach. Hence, if a symmetric AOP language was used, the benefits of the aspectual process may have been more obvious.

The *external validity* addresses the generalization of the findings.

- An important threat to the external validity of the study is the relative small size of the applications and the maintenance operations. Real-life maintenance operations may take several days or even longer. In addition, longer maintenance operations may require an iterative switching between models and code.
- Another threat is the limited scope of the maintenance operation types (e.g., no refactoring). Moreover, the chosen tasks might not be representative of real-word situations. However, we have tried to simulate "typical" and "diverse" maintenance operations given the scope of the study (addition and improvement of functionality).

- Furthermore, the results could be specific for AspectJ and Java, although they are good representative for AO and OO programming languages, respectively. The same reasoning applies to the adoption of Theme, of which the results may not be generalizeable to other AOM approaches.
- Also, the results could be specific to the chosen domains or applications. One could construct an example where AO code would not pay off and, thus, the hybrid process would be favored. In order to discard this threat, replica of this study on other applications are necessary. However, the severity of this threat appears to be low as we suspect that the above-mentioned example would possibly be non realistic (i.e., of a pedagogical nature).
- It could also be difficult to generalize the results to other kinds of aspects not used in this study.
- Finally, a possible threat is the inclusion of non-professional programmers in the experiment.

7. RELATED WORK

To our best knowledge, there are no existing empirical studies comparing and investigating the different options developers can follow starting from an aspectual design as described in section 2. However, there are a number of related works that we describe separately. We first present a number of the most prominent AOM approaches and look into their position on the matter. Then we present the existing empirical efforts in the field of AO vs. OO comparisons that are complementary to our work.

7.1 AOM approaches

In [8] Cottenier et al. present a categorization of the existing AOM approaches into two camps. The so-called "translationistic" approaches model the system so that the source code can be completely automatically generated [9, 17, 37]. As the code is complete and should not be manually enriched, the development process actually ends at the modeling level, and the choice between an aspectual or a hybrid process becomes irrelevant. Even though this approach is successfully applied in a large telecom industrial case [10], Cottenier believes that the approach would still be impractical in a large enterprise or business applications. In addition, such approaches are tightly coupled with program-level abstractions supported by current AOP languages and do not raise the abstraction level above the implementation [16].

The current high-level "elaborationistic" AOM approaches [3, 4, 25, 35] typically leave the matter of code generation and target platform outside the scope. In Theme [3], the authors shortly mention the two different options of generating towards an AO and OO platforms, and present several possible AO model-to-code mappings. However, the comparison of the two processes is out of the scope of their work. Hence, our work complements these approaches by comparing two alternative processes that are based on a state-of-the-art AOM methodology.

7.2 Existing empirical studies

Numerous recent works have explored the use of AO programming languages in software development. Hannemann and Kiczales show in [21] that AspectJ implementations of the GoF design patterns [12] improve modularity in a considerable number of cases. Garcia et al. [1] further continue on this work by providing quantitative assessment of the improvements that AspectJ platform brings to the GoF pattern implementation. Cacho et al. [5] study the aspectization of design patterns in the context of existing software systems and evaluate the interactions between pattern implementations. The benefits of exception handling modularization are investigated by Filho et al. [15] and Coelho et al. [7]. Figueiredo et al. [14] present a quantitative and qualitative study that assesses the positive and negative impacts of using an AO language in developing software product lines. Greenwood et al. [19] evaluate how AO mechanisms support enhanced incremental development and avoid early design degradation. Tonella and Ceccato [44] assess the effect of the migration of the so-called aspectizable interfaces in Java programs to AspectJ. They perform an analysis of both the internal and the external quality attributes and determine that indeed the migration to the AO platform provides externally observable benefits. Most of these empirical studies are based on a complete manual implementation of the systems. Even though some automation is used in a number of these studies, they are situated in a different context. Our work complements them by investigating the use of AO programming languages in a model-driven setting where the source code is partially automatically generated.

Munnelly et al. [29] compare the implementation of a context-aware application in AspectJ, Java and Context Toolkit [36]. According to their experiment AO improves the comprehensibility, maintainability and manageability of the source code. However, only the internal quality attributes are analyzed in their work. This paper complements their study by providing more insights in the externally observable benefits of maintaining the aspectual paradigm from models to code using Theme.

A number of studies perform a comparison of AO and OO paradigms on a modeling level. Wehrmeister et al. [45] assess how suitable each paradigm is in the context of distributed real-time and embedded systems in terms of reusability of model elements. Op de beeck et al. [31] present a quantitative study investigating the maintainability of a detailed AO and OO architecture and design. Both studies conclude AOM offers certain benefits compared to the traditional OO modeling. Once again our assessment complements these studies by investigating whether one should compose the modularized concerns using model composition techniques or keep them modularized all the way to the source code.

8. CONCLUSIONS

The aspect-oriented modeling community has implicitly been advocating the idea of reducing the role of AO artifacts at the level of implementation code. Albeit many benefits are attributed to this proposition, no empirical investigation has been performed so far to asses the effects on, e.g., the maintainability of software systems.

We have investigated whether preserving the AO paradigm throughout the system development stages offers benefits compared to the shift from AO models to OO code. Based on two applications using one of the most prominent AOM methodologies (Theme) we have conducted a number of quantitative studies and experiments. We have discovered that

the all-aspectual approach results in smaller, less complex and more modular implementation. In addition, we have demonstrated through an experimental study (with statistical significance) that in some cases the all-aspectual approach shortens the maintenance cycle. Despite the fact that the average effort metric is consistently better, the statistical results were inconclusive in the remaining cases.

Clearly, it is difficult to draw general conclusions from a single experiment involving two applications that undoubtedly focus on a specific context. However, we believe that our findings represent an important result for the AOSD research at large and stimulate the inception of a research path in this area. Indeed, we are certain that it is essential to perform such studies in order to objectively evaluate new software development techniques and paradigms.

In our future work, we intend to perform a dedicated user study in a similar context, but with focus on system understandability. We also plan on replicating this experiment in order to obtain more data points for the statistical analysis.

9. ACKNOWLEDGMENTS

This work is part of the EUREKA-ITEA EVOLVE project, and is partially funded by the Flemish government institution IWT (Institute for the Promotion of Innovation by Science and Technology in Flanders), by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

10. REFERENCES

[1] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, A. von Staa. Modularizing design patterns with aspects: A quantitative study. In *Proceedings of the 4th International Conference on AOSD*, pages 3–14. ACM, 2005.

[2] aopmetrics. http://aopmetrics.tigris.org/.

[3] E. Baniassad and S. Clarke. *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley, 2005.

[4] O. Barais, J. Klein, B. Baudry, A. Jackson, and S. Clarke. Composing multi-view aspect models. In *Proceedings of the 7th International Conference on Composition-Based Software Systems*, pages 43–52. IEEE Computer Society, 2008.

[5] N. Cacho, C. Sant'Anna, E. Figueiredo, A. Garcia, T. Batista, and C. Lucena. Composing design patterns: a scalability study of aspect-oriented programming. In *Proceedings of the 5th International Conference on AOSD*, pages 109–121. ACM, 2006.

[6] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[7] R. Coelho, A. Rashid, A. Garcia, F. C. Ferrari, N. Cacho, U. Kulesza, A. von Staa, and C. J. P. de Lucena. Assessing the impact of aspects on exception flows: An exploratory study. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, pages 207–234. Springer, 2008.

[8] T. Cottenier, A. van den Berg, and T. Elrad. Model weaving: Bridging the divide between elaborationists and translationists. In *9th International Workshop on AOM*, 2006.

[9] T. Cottenier, A. van den Berg, and T. Elrad. Joinpoint inference from behavioral specification to implementation. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, pages 476–500. Springer, 2007.

[10] T. Cottenier, A. van den Berg, and T. Elrad. The Motorola WEAVR: Model weaving in a large industrial context. In *Proceedings of the 6th International Conference on AOSD, Industrial Track*, 2007.

[11] DSG, Trinity College. Theme/UML demos. https://www.dsg.cs.tcd.ie/aspects/themeUML.

[12] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[13] European Network of Excellence on Aspect-Oriented Software Development. Toll system demonstrator. www.aosd-europe.net, 2007.

[14] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. C. Filho, and F. Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *Proceedings of the 30th International Conference on Software Engineering*, pages 261–270. ACM, 2008.

[15] F. C. Filho, N. Cacho, E. Figueiredo, a. Raquel Maranh A. Garcia, and C. M. F. Rubira. Exceptions and aspects: the devil is in the details. In *Proceedings of the 14th International Symposium on Foundations of Software Engineering*, pages 152–162. ACM, 2006.

[16] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *Proceedings of the 29th International Conference on Software Engineering*, pages 37–54. IEEE Computer Society, 2007.

[17] L. Fuentes and P. Sánchez. Execution of aspect oriented UML models. In *Proceedings of the 3rd European Conference on Model Driven Architecture Foundations and Applications*, pages 83–98. Springer, 2007.

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–355. Springer-Verlag, 2001.

[19] P. Greenwood, T. Bartolomei, E. Figueiredo, A. Garcia, N. Cacho, C. Sant'Anna, P. Borba, U. Kulesza, and A. Rashid. On the impact of aspectual decompositions on design stability: An empirical study. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, pages 176–200. Springer-Verlag, 2007.

[20] S. Hanenberg, D. Stein, and R. Unland. From aspect-oriented design to aspect-oriented programs: tool-supported translation of JPDDs into code. In *Proceedings of the 6th International Conference on AOSD*, pages 49–62. ACM, 2007.

[21] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In *Proceedings of the 17th Conference on OOPSLA*, volume 37, pages 161–173. ACM, 2002.

[22] W. H. Harrison, H. L. Ossher, P. L. Tarr, and W. Harrison. Asymmetrically vs. symmetrically organized paradigms for software composition. Technical report, Research Report RC22685, IBM Thomas J. Watson Research, 2002.

[23] A. Hovsepyan, S. Van Baelen, Y. Berbers, and W. Joosen. Generic reusable concern compositions. In *Proceedings of the 4th European Conference on Model Driven Architecture Foundations and Applications*, pages 231–245. Springer, 2008.

[24] I. Jacobson and P.-W. Ng. *Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2004.

[25] J.-M. Jézéquel. Model driven design and aspect weaving. *Software and Systems Modeling*, 2008.

[26] J. Johannes, R. Samlaus, and M. Seifert. Round-trip Support for Invasive Software Composition Systems. In *International Conference on Software Composition*. Springer-Verlag, 2009.

[27] M. M. Kandé, J. Kienzle, and A. Strohmeier. From AOP to UML - a bottom-up approach. In *Proceedings of workshop on Aspect-Oriented Modeling with UML*, 2002.

[28] M. Katara and S. Katz. Architectural views of aspects. In *Proceedings of the 2nd International Conference on AOSD*, pages 1–10. ACM, 2003.

[29] J. Munnelly, S. Fritsch, and S. Clarke. An aspect-oriented approach to the modularisation of context. In *Proceedings of the 5th IEEE International Conference on Pervasive Computing and Communications*, pages 114–124. IEEE Computer Society, 2007.

[30] NoMagic. http://www.magicdraw.com/.

[31] S. Op de beeck, D. Van Landuyt, J. Grégoire, R. Scandariato, W. Joosen, A. Jackson, and S. Clarke. Aspectual vs. component-based decomposition: a quantitative study. In *Proceedings of the 1st Workshop on Aspects in Architectural Description*, pages 1–4, March 2007.

[32] openArchitectureWare. http://www.openarchitectureware.org.

[33] A. Rashid and R. Chitchyan. Aspect-oriented requirements engineering: a roadmap. In *Proceedings of the 13th International Workshop on Software Architectures and Mobility*, pages 35–41. ACM, 2008.

[34] A. Rashid and A. Moreira. Domain models are not aspect free. In *Proceedings of the 9th International Conference On Model Driven Engineering Languages And Systems*, pages 155–169. Springer, 2006.

[35] Y. R. Reddy, S. Ghosh, R. B. France, G. Straw, J. M. Bieman, N. McEachen, E. Song, and G. Georg. Directives for composing aspect-oriented design class models. *Transactions on Aspect-Oriented Software Development*, pages 75–105, 2006.

[36] D. Salber, A. K. Dey, and G. D. Abowd. The context toolkit: aiding the development of context-enabled applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 434–441. ACM, 1999.

[37] P. Sanchez and L. Fuentes. Designing and weaving aspect-oriented executable UML models. *Journal of Object Technology*, 6:2007.

[38] C. Sant'anna, A. Garcia, C. Chavez, C. Lucena, and A. von Staa. On the reuse and maintenance of aspect-oriented software: An assessment framework. In *Proceedings of the 17th Brazilian Symposium on Software Engineering*, 2003.

[39] SINTEF. MOFScript. http://modelbased.net/mofscript/.

[40] S. Soares, P. Borba, and E. Laureano. Distribution and persistence as aspects. *Softw. Pract. Exper.*, 36(7):711–759, 2006.

[41] P. Stevens. Towards an algebraic theory of bidirectional transformations. In *Proceedings of the 4th International Conference on Graph Transformations*, pages 1–17. Springer-Verlag, 2008.

[42] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information hiding interfaces for aspect-oriented design. In *Proceedings of the 10th European Software Engineering Conference*, pages 166–175. ACM, 2005.

[43] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. S. Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119, 1999.

[44] P. Tonella and M. Ceccato. Refactoring the aspectizable interfaces: An empirical assessment. *IEEE Transactions on Software Engineering*, 31(10):819–832, 2005.

[45] M. Wehrmeister, E. P. Freitas, D. Orfanus, C. E. Pereira, and F. J. Rammig. Evaluating aspect and object-oriented concepts to model distributed embedded real-time systems using RT-UML. In *Proceedings of the 17th IFAC World Congress*, 2008.