Integrating Aspect-Orientation and Structural Annotations to Support Adaptive Middleware

Holger Mügge
University of Bonn
Bonn, Germany
muegge@cs.uni-bonn.de

Tobias Rho
University of Bonn
Bonn, Germany
rho@cs.uni-bonn.de

Armin B. Cremers
University of Bonn
Bonn, Germany
abc@cs.uni-bonn.de

ABSTRACT

To anticipate or not to anticipate — that is the question, regarding adaptive middleware in the area of ubiquitous computing. Anticipation can guarantee that both the adapted and the adapting component work together safely, but it limits the scenario space to some predictable well-known cases. This holds even more when statically typed languages are used, as we assume here. A second problem is a semantic gap between the business logic that triggers the adaptation and the technological demands of the adaptation that must be solved on the implementation level. We discuss current approaches and describe a new approach combining aspect-oriented programming with structural metadata to cope with both problems. An example illustrates how our approach will work in practice.

Categories and Subject Descriptors

C.1.3 [**Other Architecture Styles**]: Adaptable architectures; D.2.12 [**Interoperability**]: Interface definition languages

1. INTRODUCTION & GOALS

Mobile devices are widespread and become more and more powerful. We expect their capabilities to resemble today's desktop computers in the near future. What makes them really special is that they accompany their users in all everyday actions and are used in a vast variety of very different situations. Hence, context-sensitivity and dynamic adaptation to the current context are of increasing importance for software developers.

Our goals are (1) to reduce anticipation while preparing adaptive software, (2) to help bridging the semantic gap between business logic and technical solution details, and (3) to ease development of adaptive software. The solution we propose in this paper assumes that a statically typed language like Java is used.

MAI March 20, 2007 Lisbon, Portugal

2. PROBLEM ANALYSIS

2.1 Conflicting Forces

We see two main causes that make dynamic context-driven adaptation difficult. They can be represented by two pairs of conflicting forces and are visualized in figure 4 which depicts our suggestion for an overall workflow for engineering adaptive software.

Unpredictability vs. Anticipation: Developers of basic applications can not predict in what situation the users will appreciate which adaptation. Even though some adaptation use cases are well-known, the major part of potential adaptations can not be predicted, as we discussed in [7]. On the other hand, to ensure safe adaptation, both the developers of base applications and those of adaptations tend to anticipate as much as possible. Figure 4 shows this in terms of dependencies.

Technical Precision vs. Business-Logic Semantics: In order to be robust and safe, runtime adaptation of software calls for technically precise description of the involved components, e.g. in form of interface definitions. On the other hand the need for adaptation is driven by semantical relations between concepts at the business-logic level. Hence, we see a semantic gap between the triggers of adaptations (in terms of situation changes) and their realization (in terms of program code). This is shown in figure 4 as the y-axis.

2.2 An Example Scenario

In our scenario a business user visits a trade fair. He keeps a lot of documents on his mobile device and manages them by different applications.

When he enters the fair and is about to be engaged in meetings and negotiations, his device recognizes the new situation automatically and scans for adequate adaptations. The fair organization offers some special services for document management support: An *indexing* service which classifies all documents on the mobile device by associating them with the different company stands on the fair. A *Document Filter* service highlights and sorts the documents based on this classification for each application running on the device. Figure 1 gives an overview of the involved applications and services. A more detailed description is given in [7].

2.3 Requirements

The settings of context-sensitive adaptations impose the following requirements, we address in this paper.

The abstraction shared between both adapted and adapt-

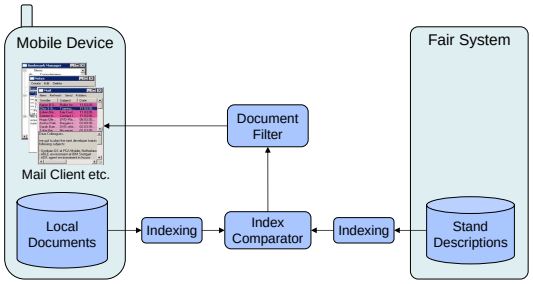

Figure 1: Adapting mobile PIM applications

ing components should support both developers. The **abstraction should be flexible enough** to allow for individual coding styles when they fulfill the functional requirements. The **additional effort for the developers should be low** and tightly integrated with their usual development environment. For the application developer it is important that he can **check, that his code adheres** to the rules of the common abstraction. It should be **precise enough** to allow the developer of adaptations to rely on it and implement the adjustment of his adaptations.

2.4 Integrating an Indexing Service

This section discusses how our approach makes a set of different document containers (notes tool and mail client) accessible for an indexing service, without anticipating this concretely in advance.

The indexing service needs access to the textual content of all documents managed by these tools. For the notes tool this is relatively easy since each stored note provides the methods getContent() and getHeading(), both returning a String-typed part of the note's content. The UML class diagramm shown in figure 2 shows the relevant part of the involved types.

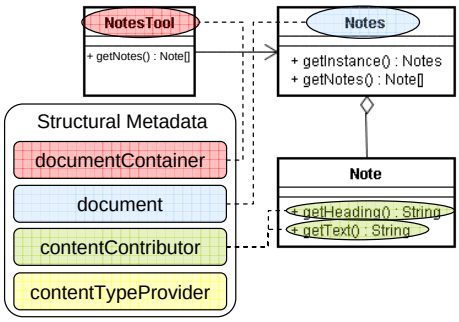

Figure 2: Accessing content of a notes tool

The class `NotesTool` which aggregates all documents is annotated with the *documentContainer* role. *contentContributor* denotes the methods returning document content. The *documentPath* role specifies via which methods the concrete documents can be reached. Analogously *contributionPath* annotates the access to *contentContributors*. Both roles are not shown in figures 2 and 3 for clarity, but listed in table 1. If the content is not of type String the *contentTypeProvider* can be used to specify the type, e. g. "text/html".

For the mail client things are more complicated. Figure 3 depicts this situation as UML class diagram.

First, the content is much more distributed: each message does not only contain the immediate content, i. e. the mail body, but holds additional textual information belonging to the content, namely the mail subject and other headers, like from, reply-to etc. Furthermore, the messages themselves are not stored in one single container. The inbox and the drafts folder behave differently and are implemented as different types. Hence, the content of all documents accessible by the mail client are distributed over objects of different classes.

Second, content is typed: mails can not only contain plain text but their content might be of different type (cf. MIME types) and may be structured (e. g. the Multipart type in the JavaMail API). Accessing the textual content therefore has to take this type into account and must be restricted to textual parts of the mail. Therefore, each *contentContributor* has associated a *contentTypeProvider*. This is given either statically as return type of the content accessing method (as in the case of the notes tool) or through an associated method providing the type dynamically.

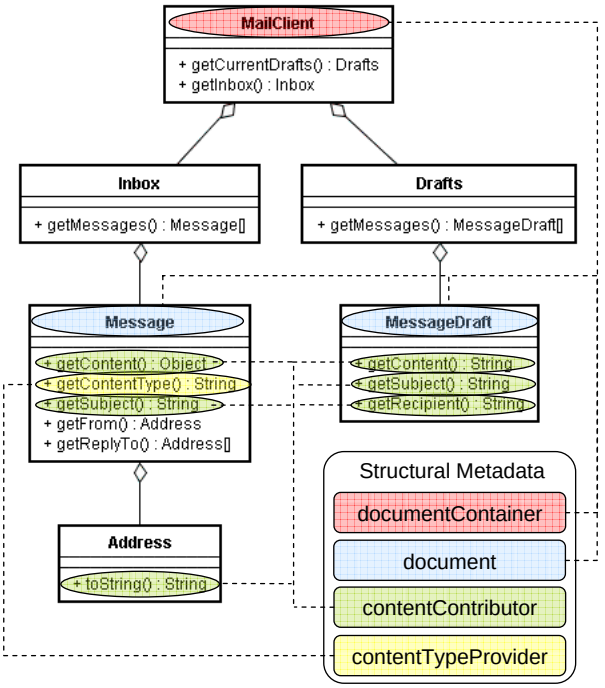

Figure 3: Accessing content of a mail client

This little variation can be moved into the structure schema itself. Thus, for each annotated content contributor that does not have associated a annotated *contentTypeProvider*, a method providing the constant type (e. g. "String" for notes) is generated at development time. Hence the base application developer does not have to deal with it. The adaptation developer does also not have to deal with these variants and can assume that for each *contentContributor* a *contentTypeProvider* is available and known.

Role	Player
documentContainer	MailClient
document	Message, MessageDraft
documentPath	Call chain
contentContributor	Message.getContent(), Message.getSubject(), MessageDraft.getContent(), etc.
contributionPath	Call chain
contentTypeProvider	Message.getContentType() (generated if not annotated)

Table 1: Roles and players for mail client content

3. PROPOSED NEW SOLUTION

In our approach the developers provide means for later adaptability by annotating semantic concepts in form of their structural realization in the software. We explain how this structural metadata is specified in section 3.1. These structural annotations are used to describe what the software actually provides, not what could be done with it, hence anticipation can be reduced significantly. Thus, direct dependency can be replaced by a weak coupling to a commonly shared structural description.

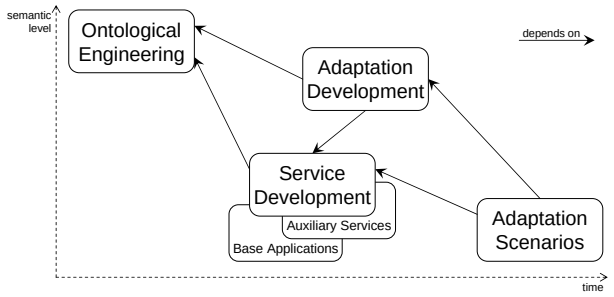

Figure 4: Proposed Workflow for Adaptation Engineering

Furthermore, structural annotations describe the adaptation points on an ontological level, i. e. they use business logic concepts for the declarations of roles and relations. This is represented in figure 4 as "Ontological Engineering" and it precedes the development of both services and adaptations (cf. "Service Development" and "Adaptation Development") in figure 4.

Adaptation developers use this conceptual information and rely on it to access and adapt base application. In section 3.2 we describe details of Logic-AJ, which is applied to expose the structure to a middleware framework. Section 3.3 presents the workflow for our example in some detail.

3.1 Structural Annotations

We accompany the source code of the base applications with structural annotations which in our example express the semantics regarding content. Therefore we define a structural schema that serves like a formally defined design pattern for content access in document containers. This schema defines and applies the following roles:

The definition of the structure schema is given as an abridged Prolog definition in figure 5 and table 1 lists all defined roles and their players for the bookmark manager.

```
schema(documentContent).

role(documentContainer).
  constraint(entity_type('type')).
  constraint(cardinality('1')).

role(document).
  constraint(entity_type('type')).

role(documentPath).
  constraint(entity_type(‘method-call')).

relation(documentContainment,
    [documentContainer, document, documentPath]).
  constraint(cardinality('1', [2], [1])).

role(contentContributor).
  constraint(entity_type('method')).

role(contributionPath).
  constraint(entity_type(‘method-call')).

relation(contentContainment,
    [document, contentContributor, contributionPath]).
  constraint(cardinality('1', [2], [3])).

role(contentTypeProvider).
  constraint(entity_type('method')).

relation(contentTyping,
    [contentContributor, contentTypeProvider).
  constraint(totality([1,2])).
```

Figure 5: Schema definition for document content

Annotating structural metadata declares roles of certain software fragment (e. g. classes, methods or calls) and provides semantic relations between them (e. g. containment, type-of). Based on these metadata the potential adaptation points[1] can be enabled. That is where AOP comes into play. Based on the actual realization of a structure an aspect can be inferred and woven into the application to expose the adaptation points as services.

3.2 LogicAJ

LogicAJ (Logic Aspects for Java) [10, 5] is an aspect-oriented extension of Java which enables generic definitions of pointcuts, advice and introductions by supporting logic meta-variables in all language constructs.

This section only sketches the main concepts of the language. A thorough introduction can be found in [10]. LogicAJ's main construct is the aspect effect shown in Figure 6. It provides generic variants of the *before*, *after* and *around* advice known from AspectJ and generic inter-type declarations of methods, fields and classes prefaced by the *introduce* keyword.

Genericity in LogicAJ is enabled by logic meta-variables which can be used uniformly in pointcuts, advice and introductions to stand in place for syntax elements like types, identifiers and parameters. The logic meta-variables share their semantics with logic variables in Prolog. Logic meta-variables are denoted syntactically by names starting with a question mark, e. g. `?entity`. Unnamed logic meta-variables are indicated by an underscore (`?_`).

[1]We use the term adaptation point here as a specific form of variation points

```
[introduce|before|after|around] <name>(<parameters>):
    <pointcut-expression>
{
  [<advice-body> | <introduction-body>]
}
```

Figure 6: Syntax of LogicAJ's generalized aspect construct.

3.3 Structural Annotations & AOP Combined

In the following we sketch the steps at development and run-time to use structural annotations and AOP in our example scenario.

The first three initial steps are independent of the development of a concrete component. They enable runtime adaptations which make reference to the described structure.

a) A structure schema for a certain concern is developed.

b) Java interfaces are modeled which represent the structure at runtime. We call these *passive structure interfaces*, because they only provide *access* to the structure.

c) Generic aspects are defined which statically weave in glue code between annotated code structure and the Java interfaces. All root nodes of the structure are exposed as service, here the classes associated with the *documentContainer* role.

d) Base application developer codes and annotates the code with metadata structures.

e) From the structural metadata additional code is generated via the generic aspects defined in the first step, that implements an interface representing the structure and provides access to the content of the document containers respectively documents.

f) Adaptation developer codes and assumes that the base application adheres to the structure, i. e. the interface is exposed as a service.

g) At runtime, the structure is utilized by runtime adaptations represented by service aspects[11]. This involves the identification of *documentContainer*(s) as part of the client software as well as *indexer* as an available service. As provided and required structure interfaces are the same applicability is assumed. The user might be asked whether she is interested in the service (composition) or not.

This paper focuses on the static part of this work flow. The definition and application of runtime adaptation aspects is thoroughly described in [7]. Figure 7 gives an overview of the whole process.

Therefore the following description will only consider step a) to e). For the first step meta data for a structure is developed that reflects the concern to be modeled. This was illustrated in section 2.4. We will now explicate the corresponding Java interfaces for the metadata and the generic aspects that are responsible for weaving runtime hooks to

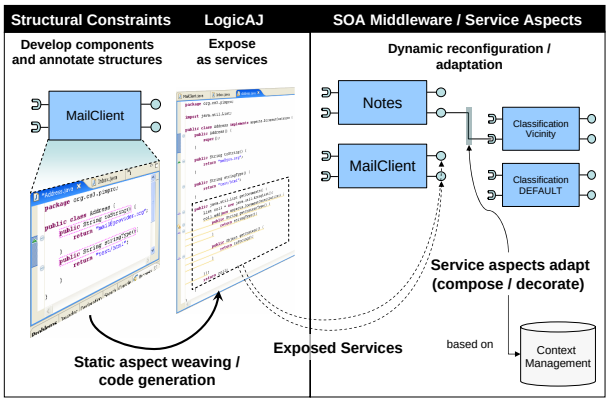

Figure 7: The complete development work flow. Metadata schemas are assumed to be developed beforehand. The left part shows the annotation step and the AOP-enabled exposure of services which provide access to the annotated structure at runtime. The right part illustrates the possible use by runtime adaptations.

these annotated code structures thereby exposing the passive structure interface.

Currently both steps have to be done manually. This is justifiable, since we consider the definition of a generic structure description the hardest part of these preparation steps. But, we are exploring ways to (semi-)automate both steps in the future (cf. section 5).

Figure 8 shows on the right side and at the bottom the Java interfaces representing the structure model. Additionally it illustrates how the annotated code structures are mapped to the Java interfaces by generic aspects. We will now describe the sub steps in more details.

(1) At first the enclosing type of every method annotated as a path to documents or document contributors is extended by the DocumentPath resp. ContributionPath interface and returns the union of all objects returned by the methods. (2) Each class with the *document* role is extended by an implementation of the *Document* interface. (3) Every method pair of the roles *documentContributor* and *contentTypeProvider* are wrapped by an implementation of the *DocumentContributor* interface and is returned by the *getContributors()* method of the enclosing *ContributionPath* or *Document* interface. (4) Every class with the *documentContainer* role is extended by the *DocumentContainer* interface. Since at least one method in the class represents a path to a document the *getDocuments()* method is already implemented. To make the *DocumentContainer* objects accessible at runtime they are exposed to the middleware as services. The Path interfaces are only implementation details and can be ignored by in the runtime adaptation developer.

Since the whole aspect implementation of the described steps is too large to be presented here we only show exemplarily in Figure 9 how step (3) is realized as an LogicAJ advice.

The pointcut *documentContribution* used by this advice represents the annotation data The around advice *getDocumentContributors* uses the pointcut *documentContribution*

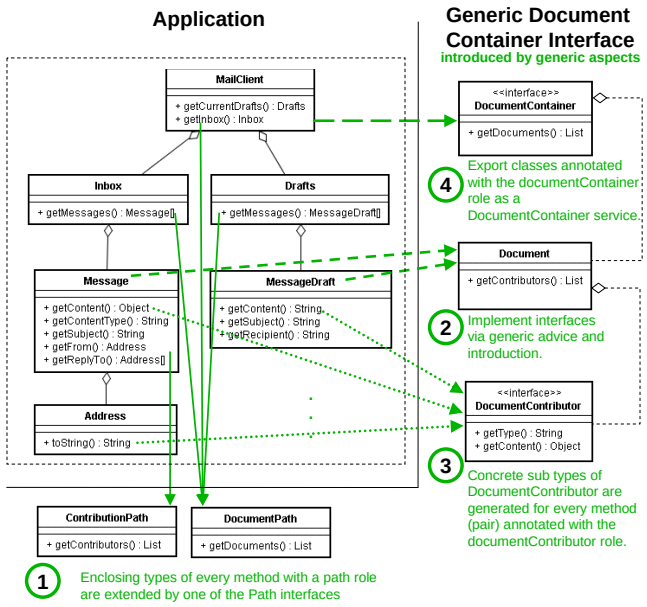

Figure 8: Applying AOP to expose the document structure to the SOA middleware.

```
1  pointcut documentContribution(?document,
2      ?contribution, ?contributionType) :
3      method(@DocumentContributor(?contributionType)
4            ?_ ?document.?contribution());
5
6  List around getDocumentContributors(?doc doc, ?contrib) :
7    documentContribution(?doc, ?contrib, ?contribType) &&
8    execution(List ?doc.getContributors()) &&
9    this(doc)
10 {
11     List coll = proceed();
12     coll.add(new DocumentContribution() {
13        public String getContentType() {
14          return doc.?contribType();
15        }
16        public Object getContent() {
17          return doc.?contrib();
18        }
19     });
20     return coll;
21 }
```

Figure 9: Generation of the DocumentContribution instances via a generic advice.

to bind types with the *document* (or *documentPath*) role to the meta variable ?document.

The pointcut *documentContribution* in line 1 binds the concrete *Document* (or *DocumentPath*) classes and contribution method names from the annotated code structure to logic meta variables. We used Java 5 annotations to represent the metadata to make the code comprehensible for developers familiar with Java/AspectJ 5[1]. The meta variables are used to intercept the execution of the *getContributors()* method of the Document or DocumentPath class (line 8) and to call the corresponding methods in the anonymous class in line 14 and 17.

Similar advice constructs address the implementation of the other interfaces parts.

Additionally, we have to care about dynamic changes in the base application. This covers the creation of *documentContainers* and changing documents or their content.

From the structural annotation a second interface is generated at development time of the base application. We call it the active structure interface, as it provides call back methods that will influence running adaptations later on.

We must ensure that the base application calls this interface appropriately. The activation code can be provided by static AOP right after the interface has been generated. For example, each instantiation of a type which is annotated as document container will call a generated addDocumentContainer() method upon the active structural interface. Analogously creation of documents or changes of their content will call specific update methods.

The calls to the active structure interface will not result in any actions before the adaptation occurs. Adaptations will register as listeners against this interface.

The developers of the adaptation have to provide glue code that reacts appropriately to events triggered by the base application. This will typically involve passive structure interfaces or other involved base applications (e. g. an indexing service).

4. RELATED WORK

To adapt software at runtime a common abstraction is needed that the base applications adhere to and the adaptations can rely on. Here we discuss briefly some basic techniques we have investigated:

Object-oriented programming provokes detailed anticipation in terms of explicit interface common for base application and adapting code. Using design patterns only supports flexibility for anticipated adaptations. The potential of context-specific indexing will be limited.

Service-Oriented programming e. g. using OSGi cf. [9] supports automation of adaptations but does not eliminate the need for their anticipation. OSGi supports annotating services with arbitrary metadata, but this is restricted to the component level.

Aspect-orientation can be used to weave the desired adaptation code to fill the gap between the indexing service and the base applications. If this is done at runtime, it can take the current context into account and provide the appropriate functionality expressed in form of a suitable advice. Most AOP approaches do not support adequate pointcut descriptions to capture join points based on context data and business-level semantics. The latter is tackled e. g. by model-based pointcuts (cf. [4]), which are based on conceptual information rather than solely on source code.

Annotations and Ontologies can in principle support interoperability of base code and adaptations. In practice the developer of both parts need to make sure that their implementations are suitable to work together. For the collection of distributed content alone this might work, but when the option of typed content or different content natures come into play, the restrictions can not be expressed in simple annotations and therefore not checked statically.

Introspection and Reflection provide in principle unlimited adaptation options at runtime. E. g. h-maps as available in the PalCom architecture (cf. [12]) reflect the complete structure of a service-oriented system even within the components. Reflective extension exist for languages like Java which originally only provide introspection (cf. [13]). We assume that complete reflection is not needed, but only certain parts of programs will be inspected or even altered at runtime in order to adjust adaptations to work with them. On the other hand for that parts of the code that are needed to fine-tune the adaptation semantic information on the business level is necessary. Hence reflection alone does not suffice.

Summarizing we propose to combine different aspects of all these approaches to tackle the requirements we elicitated in section 2.3 as we earlier suggested in [8].

5. OUTLOOK

In our approach structural metadata is mapped to an interface and an aspect in order to expose adaptation points as services. We are optimistic to automate this step by generating the according code from the structure specification. Thus the development effort would be reduced significantly.

We work on support for service retrieval and composition to partly automate finding and configuring adaptations (cf. step (g) in section 3.3). A basic retrieval model thereof is described in [6].

We are currently investigating a dynamic variant of using structural metadata. The basic idea involves a dynamic structure repository that provides reflection on the code as specified by the structures. Compared to other reflective approaches (e. g. [12]) this would lead to a quantitatively limited but qualitatively enhanced reflection. I. e. more information about but only a part of the code. Seemingly, metadata can be annotated easier (cf. step (d) in section 3.3) in some cases and we hope for a simpler way to implement the adaptive glue code.

We want to enhance structure schemata by actually using a given ontology defining terms like "document" or "title" would be of great use. We see two possible benefits: (1) this could lead to a semi-automatic detection of possible adaptations and (2) the potential of adaptivity could be exploited better, e. g. a document title could have more weight for indexing than the body.

Finally, we did not analyze yet how our approach could be used when a dynamically typed language like Smalltalk or Lisp is used. A detailed comparison to other approaches like mixin layers or ContextL as described in [3] and [2] respectively would be of great interest.

6. SUMMARY

In this paper we present a novel approach for developing adaptive service-based software. It employs structural metadata annotating selected software elements that can later be used for adaptivity. These annotations describe certain parts of the software structure on an ontological level. Based on the structural metadata, we infer aspects and weave them statically into the base code to expose adaptation points as services. At runtime we employ dynamic context-sensitive aspects to adapt the software appropriate to the current given user situation. We discuss our approach to some detail with the help of an illustrative example.

7. REFERENCES

[1] *AspectJ Compiler*. http://eclipse.org/aspectj/.

[2] P. Costanza and R. Hirschfeld. Language Constructs for Context-oriented Programming - An Overview of ContextL. Dynamic Languages Symposium, 2005.

[3] P. Costanza, R. Hirschfeld, and W. D. Meuter. Efficient Layer Activation for Switching Context-dependent Behavior. In *Joint Modular Languages Conference 2006 (JMLC2006)*. Springer LNCS, Oxford, England, 2006.

[4] A. Kellens, K. Mens, J. Brichau, and K. Gybels. Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts. In D. Thomas, editor, *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP)*, pages 501–525. Springer, LNCS 4067, 2006.

[5] G. Kniesel and T. Rho. A Definition, Overview and Taxonomy of Generic Aspect Languages. *L'Objet*, to appear, 2006.

[6] J. Kuck and M. Gnasa. Context-Sensitive Service Discovery meets Information Retrieval. In *Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2007.

[7] H. Mügge, T. Rho, D. Speicher, P. Bihler, and A. B. Cremers. Programming for Context-based Adaptability — Lessons learned about OOP, SOA, and AOP. SAKS Workshop in conjunction with GI/ITG-Tagung Kommunikation in verteilten Systemen, March 2007.

[8] H. Mügge, T. Rho, M. Winandy, M. Won, A. B. Cremers, P. Costanza, and R. Englert. Towards context-sensitive intelligence. In R. Morrison and F. Oquendo, editors, *Proceedings of European Workshop on Software Architecture*. Spinger LNCS 3527, 2005.

[9] OSGi Alliance. *OSGi Service Platform Service Compendium - Release 4*, August 2005.

[10] T. Rho and G. Kniesel. Uniform Genericity for Aspect Languages, Technical Report IAI-TR-2004-4, Computer Science Department III, University of Bonn. In *Uniform Genericity for Aspect Languages, Technical Report IAI-TR-2004-4, Computer Science Department III, University of Bonn*. Dec 2004.

[11] T. Rho, M. Schmatz, and A. B. Cremers. Towards context-sensitive service aspects, workshop on object technology for ambient intelligence and pervasive computing, in conjunction with 20th european conference on object oriented programming (ecoop 06), july 3-7, nantes, france, July 2006.

[12] U. P. Schultz, E. Corry, and K. V. Lund. Virtual Machines for Ambient Computing: A Palpable Computing Perspective. In *Workshop on Object Technology for Ambient Intelligence at ECOOP*, 2005.

[13] É. Tanter, N. Bouraqadi, and J. Noyé. Reflex – Towards an open reflective extension of Java. In A. Yonezawa and S. Matsuoka, editors, *Proc. of Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, pages 25–43. Springer, LNCS 2192, 2001.