Layered design approach for context-aware systems

Brecht Desmet, Jorge Vallejos, and Pascal Costanza
Programming Technology Laboratory
Vrije Universiteit Brussel
B-1050 Brussels, Belgium
{bdesmet, jvallejo, pascal.costanza}@vub.ac.be

Robert Hirschfeld
Hasso-Plattner-Institut
Universität Potsdam
D-14482 Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

Abstract

The omnipresent integration of computer technology in everyday applications introduces new opportunities to make software systems aware of the context in which they are used. Such context-aware systems can respond more adequately to user expectations. However, modelling the context influence inside of software systems burdens developers for several reasons. First, context-dependent behaviour might crosscut the application logic of a software system. Next, since software systems can simultaneously reside in multiple contexts, context-dependent behaviour should be composable. Furthermore, since context information is volatile, these compositions are subject to change at runtime. This paper explores how layered design approaches can be used to deal with these specific characteristics.

1. Introduction

The Ambient Intelligence vision describes scenarios in which people are pervasively surrounded by interconnected embedded and mobile devices. A pertinent issue in such a setting is the continuous changes of context in which such devices have to operate. Context-aware computing is already established as a field in which the sensing of, and reasoning about, context information is explored [9, 6, 2]. However, little attention has been paid to approaches for structuring the actual behaviour variations in context-aware systems. On the one hand, such behaviour variations are cross-cutting because in the general case, they affect different parts of the overall system. From this perspective, they give rise to a layered architecture, as conceptualized for example in feature-oriented programming, in which each behaviour variation is captured by a separate software layer. On the other hand, such context-aware layers require support for dynamic activation and deactivation due to the volatile nature of context.

The contribution of this paper is an analysis of the various issues in employing a layered architecture for context-aware systems. Specifically, we discuss:

- the emergent interactions between layers that need to coexist because a system can reside in different contexts at the same time;
- a possible formalization of such interactions;
- how these interactions affect dynamic composability of layers;
- and how the consistency of the system behaviour can be ensured in the presence of sudden context changes.

We base our discussion on an extensive example and provide a proof-of-concept implementation by combining features of ContextL, a programming language extension for Context-oriented Programming based on the notion of layers, with a logic engine using production rules for reasoning about layer interactions.

2. Example scenario

We present the software of a simplified cell phone as an illustration of a context-aware system. The application core logic of the cell phone is to ring whenever somebody calls or sends a message, and to provide the means to answer calls and read messages. Furthermore, the phone contains a list of contacts, some of them marked as VIPs.

2.1. Context-dependent adaptations

The behaviour of the application core logic can be adapted at runtime according to context changes. We introduce three *context-dependent adaptations*, each of which contains two parts: a *context condition* that explains when the adaptation is applicable and the actual *context-dependent behaviour* of the adaptation.

IgnoreAdaptation $Battery = Low \rightarrow I$
: If the battery level is low, ignore all phone calls except for contacts classified as VIP.

AnswermachineAdaptation $11pm < time < 8am \rightarrow A$
: If the time is between 11pm and 8am, activate the answering machine for incoming phone calls and the auto-reply service for messages.

RedirectAdaptation $Location = Meetingroom \rightarrow R$
: If the user is in a meeting, redirect all calls and messages to the secretary.

DiscreetAdaptation $Noise = High \rightarrow D$
: If the ambient noise is high, activate vibration and visual notifications.

CostAdaptation $Switch = On \rightarrow C$
: If the user requests so, a cost estimation for incoming phone calls is maintained. This adaptation consists of two alternatives.

- $Location = Abroad \rightarrow CI$
  If the user is abroad, *InternationalTariffAdaptation* (CI) is applicable.
- $Location = \neg Abroad \rightarrow CN$
  Otherwise, *NationalTariffAdaptation* (CN) is applicable.

2.2. User policy

Although all context conditions (battery low, time between 11pm-8am, meeting room location, noise level, and user request) can be true at the same time, the behaviour of the adaptations cannot be freely combined. This is because adaptations might interact with each other. A user policy describes what the interactions are and how they can be resolved. For instance, the following informal rules constitute a possible user policy for the cellular phone scenario.

PolicyRule I *InternationalTariffAdaptation* and *NationalTariffAdaptation* are mutually exclusive. Furthermore, both adaptations depend on *CostAdaptation* since they share some common behaviour.

PolicyRule II *IgnoreAdaptation* and *RedirectAdaptation* cannot coexist. *RedirectAdaptation* has priority.

PolicyRule III *AnswermachineAdaptation* and *RedirectAdaptation* cannot coexist. *IgnoreAdaptation* is only applied if *RedirectAdaptation* failed (e.g. the secretary did not answer the incoming phone call).

PolicyRule IV *DiscreetAdaptation* can coexist with all other adaptations without interference.

3. Problem statement

We focus on three fundamental characteristics of context-aware systems: In the general case, context-dependent behavior variations are cross-cutting, multiple behavior variations need to coexist, and they have to react to possibly unexpected context changes. Therefore, they require *modularisation*, *dynamic composability*, and means to ensure *consistency*.

3.1. Modularisation

Context-dependent behaviour can cut across multiple units of modularisation. For example, *AnswermachineAdaptation* refines the behaviour of both phone and message traffic. Without a suitable modularisation mechanism, the code of context-dependent behaviour can easily get scattered and tangled with the code of the application core logic. Such a lack in separation of concerns results in software systems that are difficult to maintain and evolve.

3.2. Dynamic composability

Since a context-aware system can simultaneously reside in multiple contexts, one needs to compose context-dependent behaviour dynamically. An important issue in the composition is caused by interactions that can occur between context-dependent behaviour. In our example, we identify the following relationship types between context-dependent behaviour: inclusion, exclusion, conditional dependency, ordering, and independence. These kind of relationships were inspired by the work of Nagy et al. [8] who used them to describe resolution strategies for aspects at shared join points.

• **Inclusion** - If the user has activated the cost estimation service and the user receives an incoming phone call while residing abroad, *InternationalTariffAdaptation* is applicable. However, according to *PolicyRule I* of Section 2.2, this adaptation cannot exist individually since it depends on *CostAdaptation*.

In this case, *CostAdaptation* is *included* in the composition whenever *InternationalTariffAdaptation* (or *NationalTariffAdaptation*) is applicable.

$$Ci, Cn \stackrel{includes}{\longrightarrow} C \qquad (1)$$

Furthermore, *PolicyRule I* describes a mutual exclusion between Ci and Cn. This is ensured by the fact that the context conditions of Ci and Cn are disjoint and complementary (the user is either abroad or not). However, it could be interesting to make such relationships explicit in order to increase program comprehension.

• **Exclusion** - If the battery level is low and the user is in a meeting, both *IgnoreAdaptation* and *RedirectAdaptation*

are applicable. These adaptations cause a semantic interaction since they ignore and redirect incoming phone calls respectively. *PolicyRule II* prescribes that phone call redirection has a higher priority, and hence the *IgnoreAdaptation* is omitted.

This means that *RedirectAdaptation excludes IgnoreAdaptation*.

$$R \overset{excludes}{\longrightarrow} I \tag{2}$$

• **Conditional dependency** - Another possible interaction arises when the user is still in the meeting room at midnight. In such a case, *AnswermachineAdaptation* and *RedirectAdaptation* are applicable. *PolicyRule III* specifies that first phone calls are redirected to the secretary (*RedirectAdaptation*). If this fails (e.g. secretary absent), the answering machine is activated (*AnswermachineAdaptation*).

In summary, the execution of *AnswermachineAdaptation depends on* the outcome of *RedirectAdaptation*.

$$A \overset{depends\ on}{\longrightarrow} R \tag{3}$$

• **Ordering** - The execution order of context-dependent behaviours, that constitute a particular composition, might influence the semantics of a context-aware system. The inclusion example (1) has an ordering constraint. Consider, for instance, that context-dependent behaviour is composed using inheritance[1]. In such a case, C should appear before Ci (or Cn) in the inheritance tree.

$$C \overset{before}{\longrightarrow} Ci, Cn \tag{4}$$

Furthermore, the conditional dependency example (3) also implicitly indicates an ordering constraint: First the *RedirectAdaptation* is tried, afterwards the *AnswermachineAdaptation* is activated.

$$R \overset{before}{\longrightarrow} A \tag{5}$$

• **Independence** - In contrast, our cell phone example also covers a case in which the order of execution does not affect the system semantics. If the ambient noise is high, the *DiscreetAdaptation* is included in the composition. This adaptation does not interact with any other adaptation in the system according to *PolicyRule IV*. Hence, there are no relationship types involved.

3.3. Consistency

We call context information volatile since it is subject to change at arbitrary moments in time. A context-aware system should adapt its behaviour according to these context changes. In this way, the possibility exists that some context condition becomes invalid while the behaviour it has triggered is still being executed. The injudicious abortion or continuation of context-dependent behaviour could lead the system into an anomalous program state. We distinguish different activation and deactivation strategies that should be applied depending on the desired semantics of the context-dependent behaviour.

• **Loyality** - Suppose there is an incoming phone call at 7:59am which means that the answering machine is activated according to *AnswermachineAdaptation*. The system should be loyal to that decision, even if the time elapsed beyond 8am. However, all subsequent incoming phone calls are not submitted to *AnswermachineAdaptation* until 11pm.

• **Promptness** - Whenever the ambient noise level exceeds a certain threshold, the *DiscreetAdaptation* should be applied immediately and vice versa. This action can be applied at arbitrary moments in time without any further conditions.

4. Layered design approach

This section provides an answer to the problems of modularization, dynamic composability, and consistency that hamper the development of context-aware systems. The solution for all problems is illustrated by means of a proof-of-concept implementation.

4.1. Modularisation

Layers in Context-oriented Programming are a good match to modularize context-dependent behaviour. In the following, we base our implementation of the cell phone scenario on ContextL, one of the first programming language extensions that explicitly support a context-oriented programming style [4]. It is an extension to the Common Lisp Object System (CLOS, [1]), which in turn is based on the notion of generic functions instead of the more widespread class-based object model. However, the context-oriented features of ContextL are conceptually independent of the CLOS object model, and a mapping of ContextL features to a hypothetical Java-style language extension called ContextJ has been described in [5].

Layers are the essential extension provided by ContextL on which all subsequent features of ContextL are based. Layers can be defined with the `deflayer` construct, for example like this.

```
(deflayer cellphone-layer)
```

Layers have a name, and partial class and method definitions can be added to them. There exists a predefined root or default layer that all definitions are automatically placed

[1] In Section 4.1 we actually compose context-dependent behaviour using inheritance.

in when they do not explicitly name a different layer. For example, consider the following interface in ContextL for making phone calls.

```
(define-layered-function accept-call (nr))
(define-layered-function receive-message (nr txt))
```

This defines two generic functions, one taking a phone number as a parameter and the other taking an additional text parameter. A default implementation to make phone calls for these as yet abstract functions can be placed in the root layer.

```
(define-layered-method accept-call (nr)
  ... phone calls inactive on this device ...)

(define-layered-method receive-message (nr txt)
  ... messages inactive on this device ...)
```

Only if the `phonecall-layer` is active, a user can actually answer phone calls and receive messages.

```
(define-layered-method accept-call
  :in-layer cellphone-layer (number)
  ... actual implementation ...)

(define-layered-method receive-message
  :in-layer cellphone-layer ()
  ... actual implementation ...)
```

Layers can be activated in the dynamic scope of a program. This layer activation is illustrated in Figure 1. Rectangular boxes represent layered functions and oval boxes represent layered methods. The latter oval boxes are contained within a larger rectangular box which denotes a layer.

```
(with-active-layers (cellphone-layer)
  ... contained code ...)
```

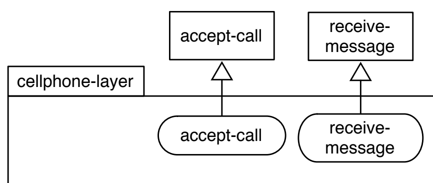

Figure 1. Layer activation.

Dynamically scoped layer activation has the effect that the layer is only active during execution of the contained code, including all the code that the contained code calls directly or indirectly. Layer activation can be nested, which means that a layer can be activated when it is already active. However, this effectively means that a layer is always active only once at a particular point in time, so nested layer activations are just ignored. This also means that on return from a dynamically scoped layer activation, a layers activity state depends on whether it was already active before or not. In other words, dynamically scoped layer activation obeys a stack-like discipline.

Likewise, layers can be deactivated with a similar `with-inactive-layers` construct that ensures that a layer is not active during the execution of some contained code, and that has no effect when that layer is already inactive. Again, on return from a dynamically scoped layer deactivation, a layers activity state depends on whether it was active before or not.

Furthermore in multithreaded Common Lisp implementations, dynamically scoped layer activation and deactivation only activates and deactivates layers for the currently running thread. If a layer is active or inactive in some other thread, it will remain so unless it is incidentally also activated or deactivated in that thread.

Multiple layers can contribute to the same layered functions. For example, the context-dependent behaviour of the *IgnoreAdaptation* (see Section 2.1) refines the behaviour of `accept-call`.

```
(deflayer ignore-layer)

(define-layered-method accept-call
  :in-layer ignore-layer (nr)
  (if (vip-p nr)          ; nr is VIP?
      (call-next-method)  ; super call
    ... ignore call ...)
```

This layered method checks whether `nr` is classified as a VIP contact. If so, a super call is invoked. Otherwise, the phone call is ignored. The activation of both `cellphone-layer` and `ignore-layer` constitute the inheritance hierarchy displayed in Figure 2.

```
(with-active-layers (cellphone-layer ignore-layer)
  ... handle phone call in context
      of low battery level ... )
```

In this particular layer composition, the super call in `accept-call` of `ignore-layer` invokes the `accept-call` of the `cellphone-layer`. In other words, if the caller is a VIP contact, we delegate the incoming phone call to the `cellphone-layer`. To the best of our knowledge, the idea of expressing designs in terms of composable layers originates from Goldstein and Bobrow [7].

Throughout this paper, we consider the existence of `answermachine-layer`, `redirect-layer`, `discreet-layer`, `cost-layer`, `int-tariff-layer`, and `nat-tariff-layer` that respectively implement the context-dependent behaviour of the *AnswermachineAdaptation*, *RedirectAdaptation*, *DiscreetAdaptation*, *CostAdaptation*, *InternationalTariffAdaptation*, and *NationalTariffAdaptation*.

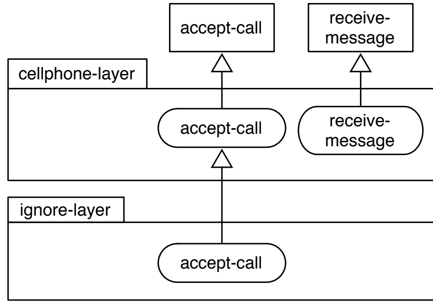

Figure 2. Composing layers in inheritance hierarchy.

4.2. Dynamic composability

Dynamic composability addresses the problem of interactions that arise in compositions of context-dependent adaptations. Our solution consists of making all these interactions explicit and available for automatic reasoning. To this end, we employ a forward-reasoning engine. The production rules of such an engine capture the context conditions of adaptations and the relationships between context-dependent adaptations.

• **Context conditions** are predicates that describe when some context-dependent behaviour is applicable. For example, Section 2.1 presents the context conditions of the cell phone scenario in both logical and textual format. We now implement these context conditions using production rules of the LISA forward chainer [10]. These production rules *complement* the ContextL layers which contain the context-dependent behaviour of the cell phone scenario.

Since the implementation of all context conditions is quite similar, we only show the implementation of the *IgnoreAdaptation* (I) context condition as an illustration. We assume the existence of a mechanism that computes high-level context information (like e.g. `(battery (level low))`) out of low-level sensor data. One could use existing tools like ContextToolkit [9], WildCAT [6], or Java Context Awareness Framework [2] to perform this job.

```
(defrule ignore-adaptation
  (battery (level low))
  =>
  (assert (layer (name ignore-layer))))
```

The body of a LISA production rule consists of a condition and an action part which are situated before and after the => symbol respectively. Logic variables are symbols with a leading question mark. For a more detailed explanation of LISA production rules, we refer to the documentation [10].

• **Composition rules** describe valid compositions of layers based on user-defined policy rules. For example, the cell phone scenario of Section 2.2 has five policy rules that informally describe what should happen in case of interacting behaviour. The formalization of these policy rules in Section 3.2 extracts five relationship types that exist among the context-dependent adaptations of the cell phone example.

As a proof of concept, we implement these relationship types (Formulas 1-5) using production rules of the LISA forward chainer.

The *inclusion* relationship is implemented as follows: If either `nat-tariff-layer` or `int-tariff-layer` is applicable (and hence defined in the fact base of LISA), `cost-layer` should be added to the fact base.

```
(defrule inclusion ()
  (or (layer (name nat-tariff-layer))
      (layer (name int-tariff-layer)))
  (not (layer (name cost-layer)))
  =>
  (assert (layer (name cost-layer))))
```

The *exclusion* relationship checks whether both `answermachine-layer` and `ignore-layer` are defined in the fact base. If so, the latter is retracted from the fact base.

```
(defrule exclusion ()
  (layer (name answermachine-layer))
  (?x (layer (name ignore-layer)))
  =>
  (retract ?x))
```

The *ordering* relationship is realized by associating a `before` slot with `layer` facts. An ordering can be established by filling in this `before` slot. The following LISA production rule implements Formula 4.

```
(defrule ordering ()
  (layer (name cost-layer))
  (?x (layer (name nat-tariff-layer)
             (not (before cost-layer))))
  =>
  (modify ?x (before cost-layer)))
```

The *conditional dependency* relationship is a special case of the ordering relationship. The `redirect-layer` is placed after the `answermachine-layer` in the inheritance hierarchy (see Figure 3).

```
(defrule conditional-dependency ()
  (layer (name redirect-layer))
  (?x (layer (name answermachine-layer)
             (not (before redirect-layer))))
  =>
  (modify ?x (before redirect-layer)))
```

The inheritance path indicates the chain of dependency. First, the layered method `accept-call` of `redirect-layer` is executed. If the redirection fails, a super call to

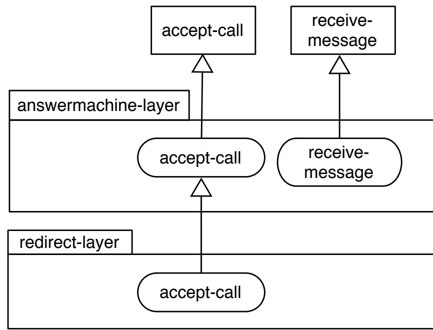

Figure 3. Conditional dependency.

answermachine-layer is invoked. Furthermore, the layered method receive-message of answermachine-layer is always invoked since redirect-layer does not refine this method.

The *independence* relationship does not interfere with other layers, thus no production rules are required.

All production rules of this section allow the LISA forward chainer to reason about the user policy and to compute valid layer compositions according to context information. Since this context information is volatile, the computation of layer compositions must happen at runtime to reflect the actual context situation. We therefore require the ability to deploy layer compositions at runtime. The with-active-layers and with-inactive-layers cannot be employed for these purposes since these constructs require manual specification of layer compositions at design time. Dynamic deployment of layer compositions can be realized with the reflective infrastructure of ContextL [3] which allows introspection and intercession of layer activations and deactivations at runtime.

The with-current-context construct is an extension to ContextL which is built entirely on top of the reflective architecture. It automatically computes layer compositions based on a user policy (implemented by means of LISA production rules) and actual context information. The with-current-context construct has, apart from computing the layer composition automatically, the same functionality as the existing with-active-layers construct: Layers are activated in a stack-like way within the dynamic scope of with-current-context, and confined to the running thread.

```
(with-current-context (... parameters ...)
  ... contained code ...)
```

The parameter list of with-current-context delimits the set of LISA production rules that are evaluated for computing the layer composition. We omit the details of this interface to improve the clarity of the example.

4.3. Consistency

Section 3.3 identifies two different strategies that describe when context-dependent adaptations can be safely activated and deactivated. We show how these strategies can be realized in ContextL.

• **Loyality** is supported by the with-current-context construct.[2] The layer composition is determined before the execution of the contained code and does not change within the dynamic scope of with-current-context[3], even if the layer composition does not reflect the actual context situation anymore. Furthermore, the layer activations (and deactivations) are confined to the running thread only.

• **Promptness** is currently not supported in ContextL. Enforcing the deactivation of context-dependent behaviour means that one jumps out of the dynamic scope of with-current-context and interrupts the execution of the contained code. This might possibly require the execution of some compensating actions (e.g. proper disconnection from a database) to avoid anomalous program behaviour. To this end, we are looking in the direction of transaction management to deal with the promptness strategy in ContextL.

The opposite situation, in which the immediate activation of context-dependent behaviour is enforced on a context change, is also not straightforwardly available in ContextL. One possible way to deal with such a requirement is to incorporate continuations. However, this idea is subject to further investigation.

5. Position statement

We advocate to implement context-aware systems using a layered design approach. The basic idea is to separate context-dependent adaptations from the basic behaviour and organize them in layers. Depending on the context in which the system resides, these layers can refine the basic behaviour. This paper proposes to use the Context-oriented Programming style of ContextL to implement the context-dependent behaviour using layers.

Next, since software systems can simultanelously reside in multiple contexts, multiple layers can be applicable which might cause interactions. We therefore explicitly describe relationships between layers. Moreover, since context information is volatile, we write these relationships in a declarative style which allows a reasoning mechanism to automatically compute valid layer compositions at runtime.

[2] This loyality strategy is inherited from the with-active-layers construct.

[3] Unless the programmer explicitly makes use of the with-inactive-layers construct to deactivate layers within a particular dynamic subscope.

Finally, we identify two strategies that prescribe when context-dependent behaviour can be safely activated or deactivated. The loyality strategy aims at consistent program behaviour by ignoring context changes within a particular scope of a software system. This strategy is explicitly available in ContextL since layer deployment is delimited to a particular dynamic scope and confined to the running thread. However, the promptness strategy, which requires immediate recomputation and deployment of layer compositions upon context changes, is not straightforwardly supported by ContextL.

6. Future work

The cell phone scenario focuses solely on five kinds of relationships between context-dependent behaviour. However, a more exhaustive study of context-aware scenarios could give rise to more interesting relationships. The same remark holds for the activation and deactivation strategies for context-dependent behaviour.

Currently, both the context conditions and relationships between context-dependent behaviour are implemented with production rules as a proof of concept. In practice, a domain-specific declarative language (e.g. XML, Prolog, or Descriptive Logics) for expressing these relationships together with a dedicated reasoning mechanism would probably be more appropriate.

The specification of relationships between context-dependent behaviour is very hard to accomplish in practice. The translation process of the user policy into declarative rules suffers from accidental underspecification, ambiguities, and contradictions. Furthermore, as the number of possible context parameters increases, one has to deal with a combinatorial explosion of possible behavioural variations. This scalability issue severely hinders the evolvability and maintainability of a context-aware system. A real solution to the problem does not exist since the combinatorial explosion is inherently present. However, we can alleviate the scalability issue by incorporating tool support for specifying relationships. The functionality of this tool support should consist of controlling the correctness of all specifications and validating the specifications against all possible context situations.

References

[1] American National Standards Institute and Information Technology Industry Council. *ANSI Common Lisp Language Specification: ANSI X3.226-1994 (R1999)*. 1999.

[2] J. E. Bardram. The Java Context Awareness Framework (JCAF) - A Service Infrastructure and Programming Framework for Context-Aware Applications. In *Pervasive*, pages 98–115, 2005.

[3] P. Costanza and R. Hirschfeld. Reflective Layer Activation in ContextL. In *Programming for Separation of Concerns (PSC) of the ACM Symposium on Applied Computing (SAC) 2007*, New York, NY, USA. ACM Press, To appear.

[4] P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming: an overview of ContextL. In *DLS '05: Proceedings of the 2005 conference on Dynamic languages symposium*, pages 1–10, New York, NY, USA, 2005. ACM Press.

[5] P. Costanza, R. Hirschfeld, and W. D. Meuter. Efficient layer activation for switching context-dependent behavior. In *JMLC '06: Proceedings of the Joint Modular Languages Conference*, volume 4228 of *Lecture Notes in Computer Science*, pages 84–103. Springer Berlin / Heidelberg, 2006.

[6] P.-C. David and T. Ledoux. Wildcat: a generic framework for context-aware applications. In *MPAC '05: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[7] I. Goldstein and D. Bobrow. A layered approach to software design. *Xerox PARC Technical Report CSL-80-5*, December 1980.

[8] I. Nagy, L. Bergmans, and M. Aksit. Composing aspects at shared join points. In *NODe '05: Proceedings of International Conference NetObjectDays*, volume P-69 of *Lecture Notes in Informatics*, Erfurt, Germany, September 2005. Gesellschaft für Informatik (GI).

[9] D. Salber, A. K. Dey, and G. D. Abowd. The context toolkit: aiding the development of context-enabled applications. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 434–441, New York, NY, USA, 1999. ACM Press.

[10] D. E. Young. *Lisp-based Intelligent Software Agents,* `http://lisa.sourceforge.net`. SourceForge, 2006.