Introducing mixin layers to support the development of context-aware systems

Brecht Desmet, Jorge Vallejos, and Pascal Costanza
Programming Technology Lab - Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussels, Belgium
{bdesmet,jvallejo,pascal.costanza}@vub.ac.be

ABSTRACT

The domain of pervasive computation introduces new opportunities to make software systems aware of the context in which they exist in order to respond more adequately to user expectations. As important as it is to have appropriate ways to obtain context information, it is to provide programmers with clean language features to model the context influence inside of software systems. This paper explores the mixin layer language construct to implement context-dependent adaptations separate from the application core logic. Together with a mechanism that dynamically composes and activates mixin layers, we argue that mixin layers possess a great potential to support the development of context-aware systems.

1. INTRODUCTION

Context-aware computing envisions scenarios in which system behaviour is parameterized by context information such as location, activity or battery level. As these context parameters change over time, context-aware systems should adapt their behaviour accordingly. Although we currently find an important amount of research on context-awareness, most of the approaches focus on the way software systems can perceive their surrounding context. We believe that this is just half-way of the development of a context-aware system. A new and totally different problem emerges when programmers start using the context information to dynamically adapt the behaviour of software systems. In this paper, we claim that mainstream language constructs to implement the contextual influence on program behaviour easily results in unmanageable designs and code scattering.

We introduce a model in which context-aware systems are described by means of adaptations. Depending on the context in which a system appears, these adaptations can be dynamically applied to the application core logic of a system. The main issue of this paper is a motivation why mixin layers are a good candidate to implement these context-dependent adaptations. We will clarify our approach with a concrete implementation of a context-aware system.

This paper is organised as follows. Section 2 presents an example of a context-aware scenario. The problems with traditional language constructs to implement context-aware scenarios are discussed in Section 3. Next, Section 4 explains how mixin layers are related to context-aware systems. Sections 5 and 6 present a concrete implementation of the example from Section 2. Additionally, Section 8 pinpoints where our approach suffers from a computational overhead and proposes a solution for that. Finally, the main ideas of this paper are summarised in Section 10.

2. MOTIVATING EXAMPLE

We present the software of a simplified cellular phone as an illustration of a context-aware system. The phone example consists of the following functionalities. First, the phone contains a list of contacts, some of them marked as VIPs. This information is encapsulated in the `contacts` class. Second, the `messages` class provides facilities to read and send messages. Third, the `journal` class keeps track of all phone traffic. Finally, the main task of the phone is to ring whenever somebody calls and to provide the means to answer calls. This functionality is offered by the `phonecalls` class. These different functionalities constitute the *application core logic* of the cellular phone.

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 contain two parts: a *context condition* that explains when the adaptation is applicable and the *actual behaviour* of the adaptation with regards to the application core logic.

IgnoreAdaptation If the battery level is low, ignore and log all phone calls except for contacts that are classified as VIPs.

AnswerMachineAdaptation If the time is between 11pm and 8am, activate the answering machine for incoming phone calls and the auto-reply service for messages.

RedirectAdaptation If the user is in the meeting room, redirect all calls and messages to the secretary.

Although the three context conditions (battery low, time between 11pm-8am and meeting room location) can all be true

at the same time, the behaviour of the adaptations cannot be freely combined. This is because adaptations might contradict each other, like e.g. *IgnoreAdaptation* and *RedirectAdaptation*. In case of a contradiction, the user can make an arbitrary decision about what should happen. For instance, in our phone example, the following set of policy rules describes the valid combinations of adaptations and how contradictions should be resolved.

PolicyRule I All adaptations can exist individually.

PolicyRule II *IgnoreAdaptation* and *AnswerMachineAdaptation* can coexist. Only VIP contacts will get in touch with the answering machine, all other contacts will be ignored. All messages receive auto-reply.

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

PolicyRule IV *AnswerMachineAdaptation* and *RedirectAdaptation* cannot coexist. *AnswerMachineAdaptation* has priority.

3. PROBLEM STATEMENT

Traditional language constructs to implement scenarios like the cellular phone example include both design patterns (e.g. strategy or decorator pattern) and conditional tests (e.g. `if` or `case`). We discuss both approaches and explain how their usage might lead to cumbersome designs.

The strategy design pattern could be a good candidate to implement the cellular phone example. In this case, we regard each possible adaptation as another strategy that deals with incoming communication. This approach does not scale since context-aware systems typically involve multiple context parameters like *BatteryLevel* = {*low, high*} or *Location* = {*meetingroom, elsewhere*}. In the worst case, a combinatorial explosion of possible behavioural variants might arise. The strategy design pattern constructs a new object for each possible behavioural variant. This approach becomes unmanageable if the number of context parameters increases.

Alternatively, one could implement the adaptations using the decorator design pattern (aka wrapper). Here, we regard the adaptations as decorators that refine the behaviour of individual objects through delegation. Since context-dependent adaptations might affect multiple classes, the developer is burdened with the implementation of non-functional behaviour that manages multiple decorators simultaneously. For example, the activation or deactivation of multiple decorators that constitute a context-dependent adaptation should be an atomic operation to ensure consistent program behaviour. Hence, the decorator design pattern is in such cases inevitably complemented with additional "bookkeeping code" which troubles the system design. This is because the decorator design pattern is actually not expressive enough to adequately implement context-dependent adaptations.

The use of conditional statements also provide no solace since the implementation of context-dependent adaptations easily leads to code scattering. Furthermore, the context information of a system, which is actually application-specific domain knowledge, is also scattered around within the test-clauses of the conditional statements. This makes it extremely difficult to extend context-aware systems. Consider for example the case in which an additional context-dependent adaptation is introduced in a context-aware system. Such an extension requires full understanding of all context information used throughout the system and might affect multiple modularisation units. Therefore, this approach is not feasible for large-scale context-aware systems.

In summary, we agree that design patterns and conditional tests are a plausible solution to implement context-aware systems. However, these traditional constructs lack expressiveness and hence induce cumbersome designs. We therefore believe that more dedicated language constructs can alleviate the design of context-aware systems substantially.

4. CONTEXT-DEPENDENT ADAPTATIONS AS MIXIN LAYERS

The claim of this paper is that the modularisation capabilities of mixin layers offer important contributions to the development of context-aware systems. The basic idea is to separate the context-dependent adaptations from the application core logic and modularise them using mixin layers. However, current practices with mixin layers are not aligned with our specific dynamic requirements in the domain of context-aware computing. Before we explain these dynamic requirements in Section 4.2, we first explain briefly what mixin layers are about in the next section.

4.1 Mixin layers

The notion of mixin layers [8] was introduced by Smaragdakis et al. as an implementation technique to support refinement of collaboration-based designs. A mixin layer is a modularisation unit that encapsulates different mixin classes each refining a single class of the collaboration. Such mixin classes (or just mixins) are also commonly known as abstract subclasses. The distinguishing feature between ordinary and abstract subclasses is that the latter have parameterized superclasses. This property enables the instantiation of mixins with various superclasses and thus supports reusability.

In practice, refinement by using mixin layers is achieved through the ability to add or specialize methods and classes. Moreover, mixin layers can also refine other mixin layers because they can be composed in an inheritance hierarchy, yielding a layered design. Furthermore, since mixin layers can affect multiple classes, they support cross-cutting modularisation to a certain degree. These properties are illustrated in Figure 1.

4.2 Dynamic requirements

Mixin layers hold some promise to implement the context-dependent adaptations because they adapt behaviour through inheritance, support cross-cutting modularisation and can be composed in an inheritance hierarchy. Unfortunately, current practices with mixin layers are not aligned with the specific characteristics of context-aware systems. This section focusses on two important issues. First, since the context information of a system is not known beforehand (e.g. location of user), we require a mechanism that *selects and composes mixin layers at runtime* based on the actual con-

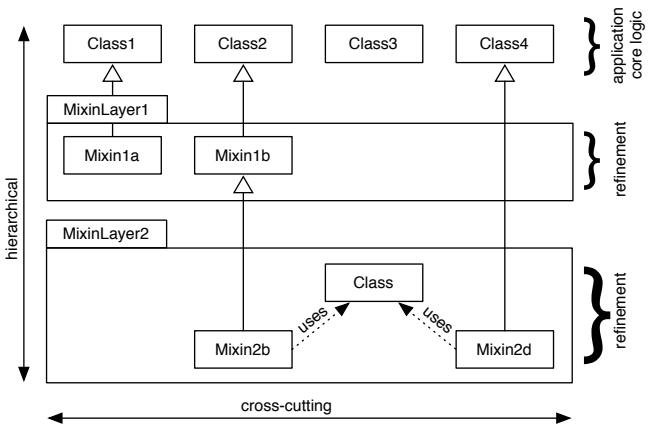

Figure 1: Mixin layers.

text information. Second, we additionally require a mechanism that *dynamically activates and deactivates mixin layers* accordingly during program execution.

Mixin layer selection and composition Our notion of this issue contrasts currentday practices with mixin layers at two levels. First, both the selection and composition of mixin layers are computed automatically based on context information. In current mixin layer implementations, this selection and composition is done manually at design time. Second, the composition of mixin layers evolves over time as the context changes. At present, compositions of mixin layers are not supposed to change at runtime and are therefore fixed at design time. Hence, we conclude that there exists a huge gap between existing practices where the composition of mixin layers does not change at runtime and the kind of dynamic composition mechanism that we require to reconfigure compositions at runtime according to context changes.

Dynamic layer activation Context-aware adaptations are accomplished by activating and deactivating mixin layers at runtime according to context changes. This pluggability can be achieved by redefining classes at runtime. Existing instances of redefined classes should be updated accordingly. On the one hand, this might look like a harsh requirement to implement in a static language like Java. On the other hand, by using the reflective capabilities of dynamic languages such as CLOS or Smalltalk, it is much more straightforward to perform class redefinitions at runtime.

The following section presents a concrete implementation of the cellular phone example from Section 2. This implementation is continued in Sections 6 and 7 that respectively discuss the mixin layer selection and composition issue and dynamic layer activation.

5. EXAMPLE IMPLEMENTATION

In an attempt to implement the cellular phone example using mixin layers, we encountered many programming languages that provide different flavours of mixin layers like CaesarJ [3], Classboxes in Smalltalk [4] and LasagneJ [9]. We chose to exploit ContextL [5] which is written in Common Lisp [2]. This implementation introduces the notion of so-called layers as an extension of the Common Lisp Object System. Our choice for this implementation is driven by the fact that ContextL supports a flexible and efficient mechanism for activating and deactivating its layers [6]. The importance of this issue is discussed in Section 7. The remainder of this section explains how the context-dependent adaptations of the cellular phone example can be implemented in ContextL.

5.1 Application core logic

We first take a closer look at the application core logic of the cellular phone. Only the classes `phonecalls` and `messages` are subject to refinement. We therefore define them as layered classes.

```
(define-layered-class phonecalls () ...)
(define-layered-class mesages () ...)
```

Layered classes provide the opportunity to define layered methods. Such methods can be refined with layers later on. In this example, the most important layered methods are `accept-call` and `receive-message` which are respectively part of the layered classes `phonecalls` and `messages`. We omit the actual implementation of the layered methods because this is not relevant here.

```
(define-layered-method accept-call
  ((p phonecalls) nr)
  ...)

(define-layered-method receive-message
  ((m messages) nr text)
  ...)
```

We additionally provide the class `cellphone` that acts as a facade for the classes of the application core logic. All incoming communication passes through the generic function `in`. This can be either a phonecall (represented by the structure `phonecall`), or a message (represented by the structure `message`). These structures contain information like caller-id, date, time, etc.

```
(defmethod in ((c cellphone) (p phonecall))
  (accept-call (get-phonecalls c) (phonecall-nr p)))

(defmethod in ((c cellphone) (m message))
  (receive-message (get-messages c) (message-nr m)
                   (message-text m)))
```

5.2 Layer definitions

We now define the `ignore-layer`, `answermachine-layer` and `redirect-layer` in ContextL for the *IgnoreAdaptation*, *AnswerMachineAdaptation* and *RedirectAdaptation* respectively.

```
(deflayer ignore-layer)
(deflayer answermachine-layer)
(deflayer redirect-layer)
```

The behavioural part of the *IgnoreAdaptation* ignores and logs phone calls from callers that are not classified as VIP. We implement this behaviour with a layered method that is part of `ignore-layer`. The latter is specified with the `:in-layer` keyword. The predicate `vip-p` checks whether the phone number `nr` is classified as a VIP contact. All ignored calls are logged in the journal with the method `add-log`.

```
(define-layered-method accept-call
  :in-layer ignore-layer ((p phonecalls) nr)
  (if (vip-p (get-contacts p) nr)
      (call-next-method)
    (add-log (get-journal p) 'ignored nr)))
```

The activation of `ignore-layer` yields the design presented in Figure 2.

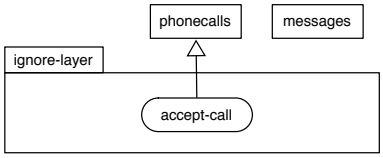

Figure 2: System design if battery level is low.

The *AnswermachineAdaptation* is cross-cutting since it affects the layered classes `phonecalls` and `messages`. The method `get-voice-message` records and returns the voice message of the caller. The method `send-message` is used to implement the auto-reply of incoming messages.

```
(define-layered-method accept-call
  :in-layer answermachine-layer ((p phonecalls) nr)
  (let ((msg (get-voice-message)))
    (add-log (get-journal p) 'auto nr msg)))

(define-layered-method receive-message
  :in-layer answermachine-layer
    ((m messages) nr text)
  (send-message m nr "I will reply asap.")
  (call-next-method))
```

Figure 3 illustrates how layers can be combined in an inheritance hierarchy. This example is an application of *PolicyRule II* that allows the combination of `ignore-layer` and `answermachine-layer`. Such composition yields the following behaviour: only callers that are classified as VIP get in touch with the answering machine. All other calls are ignored and logged in the journal. Messages always receive an auto-reply.

Finally, the implementation of the *RedirectAdaptation* is as follows. The method `redirect-call` forwards the incoming calls to the secretary `*secr*`.

```
(define-layered-method accept-call
  :in-layer redirect-layer ((p phonecalls) nr)
  (redirect-call p nr *secr*))
```

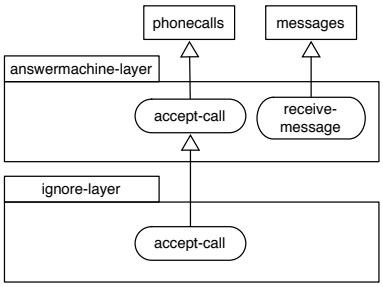

Figure 3: System design if battery level is low and time is midnight.

6. LAYER SELECTION AND COMPOSITION

The context information of context-aware systems is not known beforehand and might change over time (e.g. location of user). We therefore require a mechanism that selects and defines layer compositions automatically at run-time based on actual context information. This task is accomplished by means of a forward chainer. The production rules of such a reasoning system describe when layers are applicable and how they can be composed. We identify three categories of rules which are discussed in Sections 6.1, 6.2 and 6.3 respectively.

TransformationRules transform primitive sensor data into meaningful application-specific context information.

SelectionRules associate the conditional part with the behavioural part (i.e. layers) of an adaptation.

CompositionRules generates valid compositions of layers based on the user-defined policy rules. For example, the cellular phone scenario of Section 2 has four policy rules that precisely describe what should happen if multiple layers coexist.

We employ an efficient forward chainer called LISA [12] to perform the reasoning job.

6.1 Deducing high-level context information

The following *TransformationRules* deduce meaningful context information for the application (e.g. `(battery (level low))`) out of low-level sensor information (e.g. `(battery (percentage 15))`). LISA provides the facilities to group relevant rules in so-called rule-contexts. For example, the following rules are all part of the rule-context `"in"` which stands for incoming communication. The body of a LISA-rule consists of a condition and an action part which are situated before and after the `=>` symbol respectively. Variables are indicated with a leading question mark.

```
(defrule battery-low (:context "in")
  (?inst (battery (percentage ?x (< ?x 20))
                  (level (not low))))
  =>
  (modify ?inst (level low)))

(defrule time-night (:context "in")
  (?inst (time-of-day (hour ?x (or (>= ?x 23)
                                   (< ?x 8)))
                      (period (not night))))
  =>
  (modify ?inst (period night)))

(defrule location-meeting (:context "in")
  (?inst (location (longitude 5) (latitude 13)
                   (place (not meeting))))
  =>
  (modify ?inst (place meeting)))
```

This is only a light-weight approach in comparison with contemporary context modelling and reasoning tools like [10] and [11]. Since our major concern is language design for context-aware systems, we intentionally do not devote special attention to the modelling issue.

6.2 Layer selection

We are now able to decide which mixin layers should be activated according to the current (high-level) context information. This is done by the *SelectionRules* which are also grouped in the rule-context `"in"`. They represent the adhesive means between the context conditions on the one hand and the context-dependent behaviours (i.e. layers) on the other hand.

```
(defrule ignore-adaptation (:context "in")
  (battery (level low))
  =>
  (assert (layer (name ignore-layer))))

(defrule answermachine-layer (:context "in")
  (time-of-day (period night))
  =>
  (assert (layer (name answermachine-layer))))

(defrule redirect-layer (:context "in")
  (location (place meeting))
  =>
  (assert (layer (name redirect-layer))))
```

6.3 Composing adaptations

The final step consists of composing the selected layers of the previous step in a linear inheritance hierarchy. We therefore added a `before` slot to each `layer` fact in LISA. In this way, we can unambiguously define the order between the layers using *CompositionRules* which implement the policy rules of the cellular phone example. For instance, the following LISA rule `compose-2` implements *PolicyRule II*. It says that, if both `ignore-layer` and `answermachine-layer` are selected, `answermachine-layer` should appear before `ignore-layer`.

```
(defrule compose-2 (:context "in")
  (layer (name answermachine-layer))
  (?y (layer (name ignore-layer)
             (before (not answermachine-layer))))
  =>
  (modify ?y (before answermachine-layer)))
```

PolicyRule IV gives priority to `answermachine-layer` if both `redirect-layer` and `answermachine-layer` coexist. This is implemented as follows.

```
(defrule compose-4 (:context "in")
  (?x (layer (name redirect-layer)))
  (layer (name answermachine-layer))
  =>
  (retract ?x))
```

7. DYNAMIC LAYER ACTIVATION

Once the appropriate layer selection and composition is determined by the reasoning system, we need to activate these layers dynamically. ContextL supports an efficient mechanism to activate layers at runtime within a dynamic scope. Unfortunately, the current implementation requires that the layer composition is specified manually at design time. We therefore extend ContextL with a construct that computes the layer composition automatically using the LISA rules from Section 6. For example, a context-aware incoming phone call looks as follows.

```
(with-current-context ("in")
  (in *cellphone* *phonecall*))
```

The `with-current-context` macro carries out the following steps. First, low-level sensor information is added to the facts database of LISA. Next, the forward chainer evaluates all rules of the rule-context `"in"`. The parameter list of `with-current-context` might contain an arbitrary number of rule-contexts. The result of the reasoning process is a list of layers that describes the linear inheritance hierarchy. Finally, these layers are applied to all objects of the current thread within the dynamic scope of `with-current-context`. This approach ensures that all method calls within the boundary of the dynamic scope will be subject to the active layers and all other method calls remain unaffected. We therefore consider dynamic scoping in this case as a powerful means to ensure consistent program behaviour.

8. LAZY LAYER ACTIVATION

The current implementation of `with-current-context` suffers from a substantial overhead in cases where some layers are activated that are not relevant within the dynamic scope of `with-current-context`. We illustrate the problem with a simple example.

8.1 Extended cellular phone example

We extend the functionality of the cellular phone from Section 2 to support multiple connection types for outgoing communication: wifi, bluetooth, and default mobile connection. The connection types bluetooth and wifi are context-dependent adaptations because they are only applicable if there is a wifi and/or bluetooth connection in the surrounding environment. If the user is trying to make a phone call while there is a wifi connection available, the system will first try to perform the phone call via VoIP. If this fails (e.g. the receiver is not online), the system will use the mobile connection instead. The cellular phone also supports message sending. In this case, the system will try bluetooth and/or wifi connection to send messages before using the

mobile connection. The main difference between the outgoing phone call and message is that outgoing phone calls only check for wifi connection, whereas outgoing messages check for bluetooth and/or wifi connection. This policy is followed because the bluetooth connection type is not suited for phone traffic.

The application core logic of the cellular phone always employs the default mobile connection for making phone calls or sending messages. In addition, we define two mixin layers `bluetooth-layer` and `wifi-layer` that implement the context-dependent behaviour of bluetooth and wifi connection type respectively. These layers are only applicable if their signal level is higher than 20 percent. The user policy states that if both bluetooth and wifi connection type are available, wifi connection type is tried first. All this is shown in the following *Selection- and CompositionRules*.

```
(defrule bluetooth (:context "out")
  (bluetooth (signal ?x (> ?x 20)))
  =>
  (assert (layer (name bluetooth-layer))))

(defrule wifi (:context "out")
  (wifi (signal ?x (> ?x 20)))
  =>
  (assert (layer (name wifi-layer))))

(defrule compose (:context "out")
  (layer (name bluetooth-layer))
  (?x (layer (name wifi-layer)))
  =>
  (modify ?y (before bluetooth-layer)))
```

The following functions `perform-call` and `send-message`, which are respectively part of the layered classes `phonecalls` and `messages`, deal with the outgoing communication of the cellular phone. (This example does not deal with the actual implementation of performing phone calls or sending messages. Instead, we just print some informative sentences on the screen.)

```
(define-layered-method perform-call
  ((p phonecalls) nr)
  (print "phone call via mobile network"))

(define-layered-method send-message
  ((m messages) nr text)
  (print "message via mobile network"))
```

Next, we implement the `wifi-layer`. This layer attempts to employ the available wifi network to make phone calls or sending messages. We assume the existence of the method `contact-available-p` that checks whether the receiver is available via wifi connection type.

```
(deflayer wifi-layer ()
  ((connection :accessor wifi-connection)))

(define-layered-method perform-call
  :in-layer wifi-layer ((p phonecalls) nr)
  (if (contact-available-p
       (wifi-connection
        (find-layer 'wifi-layer)) nr)
      (print "performing phone call via wifi")
    (call-next-method)))

(define-layered-method send-message
  :in-layer wifi-layer ((m messages) nr text)
  (if (contact-available-p
       (wifi-connection
        (find-layer 'wifi-layer)) nr)
      (print "sending message via wifi")
    (call-next-method)))
```

The `bluetooth-layer` looks very similar, but is limited to messages.

```
(deflayer bluetooth-layer ()
  ((connection :accessor bluetooth-connection)))

(define-layered-method send-message
  :in-layer bluetooth-layer ((m messages) nr text)
  (if (contact-available-p
       (bluetooth-connection
        (find-layer 'bluetooth-layer)) nr)
      (print "sending message via bluetooth")
    (call-next-method)))
```

Similar to the generic function `in` that deals with all incoming communication, we implement `out` that deals with all outgoing communication (`phonecall` or `message`).

```
(defmethod out ((c cellphone) (p phonecall))
  (perform-call (get-phonecalls c)
                (phonecall-nr p)))

(defmethod out ((c cellphone) (m message))
  (send-message (get-messages c) (message-nr m)
                (message-text m)))
```

Finally, the code for making phone calls or sending messages looks as follows. The variable `*phone*` is an instance of the class `cellphone` which represents the cellular phone system. The variable `*x*` can be an object of type `phonecall` or `message`. The generic function `out` takes care of the type dispatching.

```
(with-current-context ("out")
  (out *phone* *x*))
```

8.2 Efficiency issue

The system always needs to check and possibly activate the `bluetooth-layer` or `wifi-layer`. Although, if `*x*` is of type

phonecall, the bluetooth-layer is not of interest. Hence, this approach suffers from a potential overhead. One might suggest to overcome this problem by placing the layer activations in the respective method specializers and putting the wifi and bluetooth production rules in different rule-contexts. The following modified code shows what this alternative approach looks like.

```
45 (defrule bluetooth (:context "bluetooth-out") ...)
46 (defrule wifi (:context "wifi-out") ...)
47
48 (defmethod out ((c cellphone) (p phonecall))
49   (with-current-context ("wifi-out")
50     ...))
51
52 (defmethod out ((c cellphone) (m message))
53   (with-current-context ("bluetooth-out" "wifi-out")
54     ...))
55
56 (out *phone* *x*)
```

This approach is not desirable because of the following two reasons. First, the programmer is responsible for doing some abstract interpretation about the source code in order to decide where and which rule-contexts should be activated to avoid overhead. Depending on the complexity and the size of the context-aware system, this job can become unmanageable. Second, from a language engineering perspective, it is not advised to pollute code with scattered layer activations (line 49 and 53) for avoiding overhead only. The pollution makes the code difficult to comprehend and extend because there are implicit relationships between the scattered layer activations. These implicit relationships become visible in cases where we, for example, want to deactivate the context-awareness of the outgoing communication.

8.3 Delaying layer activation

The essence of the previous problem is that the selection, composition and activation of layers is accomplished before the body of with-current-context is executed. This can be solved by delaying the layer activations until it is required. We call this approach *lazy layer activation* inspired by the notion of lazy evaluation.

We explain the semantics of this approach using the extended cellular phone example from Section 8.1. Consider for example that the variable *x* is of type message in line 44. The layer selection, composition and activation is delayed until the layered method send-message is called in line 41. This method can be adapted with both the wifi-layer (lines 19-25) and bluetooth-layer (lines 29-35). We employ the LISA rules to determine whether these mixin layers should be activated according to the current context information. Let us assume that there is only a bluetooth connection available in the surrounding environment. From now on, the bluetooth-layer remains active for all subsequent method calls within the dynamic scope of with-current-context (line 43).

The delayed layer selection requires a backward chainer to avoid computational overhead of the reasoning system. For example, when the control flow reaches the layered method send-message in line 41, there are only two candidate layers bluetooth-layer and wifi-layer. A backward chainer determines the applicability of these candidate layers by trying to satisfy their context conditions. The way how the layer composition is computed remains unchanged, a forward chainer is still the ideal candidate for this task.

9. RELATED WORK

This section discusses two approaches that deal with dynamic adaptation of system behaviour and briefly explains the differences with our approach.

Amano et al. propose the *LEAD++* description language [1] to support dynamic adaptability of software systems driven by changing runtime environments (i.e. context information). To this end, LEAD++ introduces the mechanism of so-called *adaptable procedures* and adaptable methods which are very similar to *generic functions* and methods in the Common Lisp Object System (CLOS). The main difference between both is that adaptable procedures have a more fine-grained method dispatch. LEAD++ contrasts our approach in the following ways. First, adaptable procedures have no explicit support to deal with cross-cutting modularisation. Next, the dispatch conditions are written in an operational style and reside in the adaptable method definitions. The latter design choice makes it hard to make collaborative decisions among multiple adaptable procedures. On the contrary, we distinctly separate the conditional (i.e. declarative rules) and behavioural (i.e. mixin layers) part of adaptations in order to support reusability, extensibility, and cooperation. Finally, the behavioural variations in LEAD++ are realised by means of reflection whereas we employ mixin composition.

The *Chisel* dynamic adaptation framework [7] of Keeney et al. addresses the problem of unanticipated policy-driven, context-aware dynamic adaptation of software systems. The authors promote the use of a meta-object protocol (MOP) to accomplish unanticipated dynamic adaptations. More concretely, they developed a so-called *meta-level adaptation manager* that performs run-time switches between metatypes of (base level) service objects in response to events. The decisions of this adaptation manager are guided by the Chisel policy language. The latter associates metatypes of service objects with run-time events (i.e. relevant context changes) using logical rules. This approach suffers from the following issues. First, the computational cost of continuously polling, reasoning and switching between metatypes at run-time is unclear. Consider the case in which Chisel continuously switches the metatype of a seldom used network protocol service object. In our approach, we limit the context acquisition and reasoning to well-specified places in the source code. Second, the policy language has no disciplined way to control the scope of the adaptations. For example, with regards to our cellular phone illustration, once the decision has been made of how to respond to a phone call, the system should be loyal to that decision until the phone call finishes. Such functionality cannot be implemented in the Chisel framework without workarounds like e.g. manually adding locks. In contrast, we confine adaptations to dynamic scopes in the program execution. Finally, there is no explicit support to express relationships between metatypes

like e.g. coexistence or mutual exclusion.

10. POSITION STATEMENT

This paper envisions context-aware scenarios in terms of context-dependent adaptations that can be applied to the application core logic of a system. Since traditional implementation techniques like design patterns and conditional statements easily lead to cumbersome designs, we advocate that context-aware systems require more dedicated language constructs. To this end, we introduce mixin layers as a suitable candidate to implement context-dependent adaptations while keeping the overall design lucid.

We have several reasons to believe that mixin layers provide better means in comparison with traditional approaches to develop context-aware systems. First, mixin layers go beyond the modularisation capabilities of design patterns since they support cross-cutting modularisation to a certain degree. Second, mixin layers are singletons and their application to objects do not require object instantiation. Third, mixin layers are not confined to a static inheritance hierarchy because the mixins have no fixed superclasses. All these characteristics enable the developer to focus on the implementation of the application logic without suffering from integrating "bookkeeping code" in the design to deal with context-aware issues.

Another important property of our approach is that context information is explicitly available in the system as declarative rules. In this way, the mixin layer selection and composition can be computed efficiently using a reasoning system. Moreover, the centralised context information considerably supports program comprehension and extensibility. In addition, we propose a more advanced layer selection and composition mechanism, called lazy layer activation, that reduces computational overhead of the reasoning system.

11. REFERENCES

[1] N. Amano and T. Watanabe. Lead++: An object-oriented reflective language for dynamically adaptable software model. *IEICE Trans. Fundamentals*, E82-A(6):1009–1016, 1999.

[2] American National Standards Institute and Information Technology Industry Council. *American National Standard for information technology: programming language — Common LISP: ANSI X3.226-1994*. 1996.

[3] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. Overview of caesarj. *Transactions on AOSD I, LNCS*, 3880:135 – 173, 2006.

[4] A. Bergel, S. Ducasse, and R. Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Proceedings of the Joint Modular Languages Conference*, volume 2789, pages 122–131. Springer-Verlag, 2003.

[5] P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming - an overview of contextl. *Dynamic Languages Symposium, co-located with OOPSLA*, 2005.

[6] P. Costanza, R. Hirschfeld, and W. D. Meuter. Efficient layer activation for switching context-dependent behavior. In *Proceedings of the Joint Modular Languages Conference (to appear)*. Springer LNCS, 2006.

[7] J. Keeney and V. Cahill. Chisel: A policy-driven, context-aware, dynamic adaptation framework. In *Proceedings of the Fourth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003)*, pages 3–14, 2003.

[8] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 550–570. Springer-Verlag LNCS 1445, 1998.

[9] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. N. Jørgensen. Dynamic and selective combination of extensions in component-based applications. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE'01)*, pages 233–242. IEEE Computer Society, May 12–19 2001.

[10] A.-Y. Turhan, T. Springer, and M. Berger. Pushing doors for modeling contexts with owl dl a case study. In *PERCOMW '06: Proceedings of the Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops*, page 13, Washington, DC, USA, 2006. IEEE Computer Society.

[11] D. Wagelaar and V. Jonckers. Explicit platform models for mda. In *MoDELS*, pages 367–381, 2005.

[12] D. E. Young. Lisp-based intelligent software agents `http://lisa.sourceforge.net`, 2006.