Highly Dynamic Behaviour Adaptability through Prototypes with Subjective Multimethods

Sebastián González
Département d'Ingénierie Informatique
Université catholique de Louvain
1348 Louvain-la-Neuve
Belgium
s.gonzalez@uclouvain.be

Kim Mens
Département d'Ingénierie Informatique
Université catholique de Louvain
1348 Louvain-la-Neuve
Belgium
kim.mens@uclouvain.be

Patrick Heymans
Faculté d'Informatique
University of Namur
5000 Namur
Belgium
phe@info.fundp.ac.be

Abstract

With the advent of ambient intelligence and advances in mobile hardware technology, the next generation of software systems will require the ability to gracefully and dynamically adapt to changes in their surrounding environment. Contemporary languages provide no dedicated support to this end, thus requiring software developers to achieve this run-time adaptability through the use of specific design patterns and architectural solutions. As a consequence, all possible variability points of mobile systems need to be anticipated up front. Instead, we aim at solving the problem at the language level. We propose a new programming language called Ambience that provides dedicated language mechanisms to manage changing contexts and deal with run-time adaptation of mobile applications to those contexts. The language abstractions we propose are based on a prototype-based programming model that features multimethods and subjective object behaviour. We illustrate and motivate our approach by means of two running examples that were implemented in our language.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages

Keywords Context-oriented programming, subjective dispatch, multiple dispatch, prototype-based programming, ambient intelligence

1. Introduction

Our research starts from the premise that, due to the particular characteristics of mobile systems, contemporary programming languages fall short in providing adequate abstractions for programming such systems. We are therefore conducting research on programming languages and models that provide better support for building mobile systems, and concentrate in particular on the ability of those systems to dynamically adapt their behaviour to changing contexts.

Mobile systems are interactive systems in the large. Thanks to their physical autonomy (own power source and wireless connectivity), mobile computers can freely enter and leave the open networks they encounter and engage in communication with other fixed or mobile devices that are part of the same network. Applications running on mobile systems should be aware of their execution context[1] and should adapt dynamically to such context so that they can provide a service that fulfils the user needs to the best extent possible. As mobile computing progresses towards this vision, full dynamic software adaptation to the context becomes increasingly important: the capability of a program to respond to changes that occur in its operating environment through the dynamic transformation and reconfiguration of its components and services.

Context-aware dynamic software variability is key to the construction of applications that are smart with respect to the user needs and adaptable to the current environment. We call such applications *ambient-oriented*, a term derived from Ambient Intelligence [13]. Ambient-oriented applications question the underlying assumption that a single application behaviour can be articulated and anticipated completely, and replace it with the view that application behaviour should

[1] Here *context* is used in a broad sense: people and objects in the vicinity, environmental properties such as lighting and noise, device status such as battery charge and network signal strength, available network peers and the services they offer, and so on.

be causally connected to its context and so flexible as to gracefully accommodate the most varied circumstances [8].

Using current programming technologies, run-time adaptability is often a design aspect derived from the software architecture. For instance, the Factory design pattern [10] allows the introduction of a certain degree of variability in the composition of software, by letting third-party code become an active part of an application. The State pattern allows applications to change their behaviour at run-time by reconfiguring the collaborations among their components (objects). More involved mechanisms can be built on top of these basic techniques, most notably the plug-in architectures of many large-scale applications. Unfortunately, in all existing techniques variability points are fixed by design, and little has been achieved regarding the interplay between context-awareness and dynamic software variability.

Instead of investigating advanced software architectures or intricate design patterns to enhance the dynamic adaptability of mobile systems, we try to solve the problem at the level of the programming language. More precisely, we investigate adequate programming language abstractions that would render context-aware, self-adaptable mobile applications easier to develop. To test out our ideas we designed and implemented a new language called *Ambience*, geared towards ambient intelligence. This language is strongly inspired by other similar languages, but, given that we implemented it ourselves, serves as our research vehicle in which to experiment with novel language abstractions. In this paper we focus on the notion of subjective multimethods to achieve run-time behaviour adaptability of mobile systems to changes in their surrounding context.

Rather than explaining the full language syntax and semantics, in this paper we opted for a more example-driven explanation.[2] Section 2 starts with an example of a typical ambient-oriented application and states the basic requirements of software adaptability and context-awareness that we aim to tackle. Section 3 then introduces our basic behavioural adaptation mechanism, explaining how we reify contextual information and illustrating how this information can influence object behaviour. Section 4 presents an initial set of techniques we have devised to manage this context information in a coherent way, avoiding errors that can arise from the concurrent (real-time) modification of the reification. Sections 5 discusses the qualities and rough edges of our approach. Section 6 presents the existing language-based approaches that are similar to ours. Finally, before concluding, Section 7 describes some research directions that require further exploration.

[2] A full description of the language and its features will be the subject of a forthcoming paper. Nevertheless, all examples given in this paper have actually been implemented in the current version of the language.

2. Ambient-oriented applications

The software that runs on modern mobile devices is hardly able to cooperate with its environment in a service-oriented fashion. Applications exhibit fixed functionality and fixed communication patterns, for example an agenda running on a smartphone that synchronises with a desktop computer on demand, or the technical support service of a company that schedules the visits of mobile technicians. Applications seldom exploit service discovery, let alone adapt their services to the context. For this reason, this section presents a scenario that serves as motivation and illustrates the kind of advanced collaborations we wish to enable with our approach. In Section 4 we will show how we implemented this example in Ambience. After having explained the scenario, the set of requirements we set forth for ambient-oriented applications is given explicitly.

2.1 Scenario: smartphone and GPS integration

The following simple scenario illustrates the relevance of dynamic context-aware behaviour adaptation in a typical ambient-oriented application. Ambient-oriented programming languages should be able to deal naturally with such a scenario:

> The CityMaps application for smartphones contains static maps of cities, annotated with information such as street names and special spots (hospitals, hotels, public transportation stops), much like the maps one could find in a tourist book or on Google Maps.[3] Although the CityMaps service is useful on its own, modern users expect more dynamic features. The ACME company has detected such expectation and has developed a GPS hardware module for smartphones. Once connected, the module enhances the functionality of CityMaps such that the map section drawn on the screen is updated in real-time according to the current location of the user, detected through GPS. Additionally, the user's avatar is drawn at the right spot of the map. The net effect of connecting the GPS module to the smartphone is that the CityMaps application can be used in a more navigational fashion. When the module is disconnected, the application reverts to its default static behaviour.

Instead of a hardware module, an alternative scenario would be that the GPS service were provided wirelessly by an onboard GPS system in a car. The actual details concerning how a service is acquired (e.g. from the network or by physically plugging it) are irrelevant to the point we want to make. We consider all such scenarios equivalent. The important point is that application behaviour can be improved or degraded dynamically according to the services found in the environment.

[3] See http://maps.google.com/

2.2 Key requirements of ambient applications

We now present a small list of characteristics we consider essential for ambient-oriented applications. With this list we wish to contribute to the notion of *ambient-oriented programming* [6], so that it not only encompasses language features that are useful in dealing with the hardware phenomena observed in mobile computing (e.g. volatile connections, inherent concurrency, peer-to-peer communication), but also language features that differentiate ambient-oriented applications from conventional distributed applications at the application level. As stated previously, ambient-oriented applications should be "smart" with respect to the user needs and the surrounding environment. To this end, context-awareness and dynamic behaviour adaptability should be natural features of ambient-oriented applications. This general goal is expressed by the following requirements:

1. Applications should share vocabularies that permit their interaction. Without a common understanding of a given domain, communication, and therefore cooperation, are impossible.
2. Application behaviour should be coherent with respect to the current context. Coherency mainly amounts to accuracy and timeliness as follows:
   (a) **Context detection** The detection of the changing real-world context has to be timely and accurate, whether such detection is performed by means of sensors, by analysis of network information, or indirectly by inference mechanisms based on perceived data.
   (b) **Context representation** The computer representation of the context should facilitate rapid context updates so that perceived changes are quickly incorporated into the representation (timeliness). Furthermore, such representation should be able to keep the context information that is relevant to an application (accuracy).
   (c) **Context effect** The link between the context representation and application behaviour should be so direct that context changes effectively reflect on relevant parts of application behaviour (accuracy), and do so promptly (timeliness). On the other hand, context changes should not affect unrelated parts of applications (again accuracy).

   Failure to support the previous three requirements will result in discrepancies between user expectations and the behaviour actually exhibited by applications.

All requirements above are fulfilled by the approach presented in this paper, except for context detection (2a), which we have left out of the scope of our research. We assume in the remainder that environment changes are detected accurately by some means, and are advertised timely to the run-time system.

3. A subjective approach to context adaptation

To meet the requirements of ambient-oriented applications described in Section 2.2, we have developed a proof of concept language called Ambience.[4] Ambience is a dynamically typed, prototype-based language with delegation-based multiple inheritance. Since every first-class program entity is an object, and all interaction among objects takes place through message passing, the model is *purely* object-based. As a frame of reference, all these features are shared by the Self language [18]. Additionally, Ambience features symmetric multimethods [9] and subjective dispatch [16]. This section starts by explaining these language features, which are core to our approach; the remainder of the section explains the way we exploit them to achieve dynamic behavioural adaptation to the context, illustrated by examples in Ambience.

3.1 Prototypes with Multiple Dispatch

Ambience borrows the Prototypes with Multiple Dispatch computation model from Slate [16] and is also inspired by the similar object system of Cecil [2]. Multiple dispatch departs from the idea that messages are passed to a single distinct receiver. A more expressive form of message passing is obtained where all arguments participate in method lookup. A method that takes advantage of such a multiple dispatch mechanism is called a *multimethod*.

The definition of a multimethod specifies the kind of arguments for which the method is designed to work. To this end, each formal argument declaration is annotated with an *argument specialiser*. In Ambience, argument specialisers are plain objects.[5] Given a message with a selector and list of actual arguments, a multimethod is said to be *applicable* for that specific message if (1) the message selector matches the method selector and (2) each argument specialiser can be found in the delegation graph of the corresponding message argument. As an example consider two objects, one representing a prototypical mobile phone (`mobile-phone`) and another representing a prototypical phone call (`phone-call`). A method that handles incoming calls can be defined this way:

```
receive: call (phone-call) on: phone (mobile-phone)
[ activate: phone ringtone during: 20 seconds.
  enqueue: call in: phone incoming-calls ]
```

As can be observed, the syntax of Ambience is derived from the syntax of Smalltalk [11], meant to resemble plain English. The method selector is `receive:on:`, the formal argument names are `call` and `phone`, and the `phone-call` and `mobile-phone` prototypes are the argument specialis-

[4] A prototype implementation of Ambience in Common Lisp is available at `http://www.info.ucl.ac.be/~sgm/ambience.html`

[5] In contrast with the multimethods of class-based languages such as MultiJava [3], which use classes as argument specialisers.

ers.[6] The method implementation is written as a code block between square brackets.

Multimethods in Ambience are said to be *symmetric* because all arguments share the same status in the interaction described by the method [9]. There is no distinguished receiver object, and therefore also no special `this` or `self` keywords; all participants have explicit names in the method signature. In contrast with asymmetric methods which are defined inside the class or object to which they belong, symmetric multimethods are defined outside objects, as illustrated by the previous example. Symmetric multimethods do not belong to one single object; rather, they belong simultaneously to each specialiser [16].

Another consequence of symmetry is the natural support of "receiverless" methods and messages. These start syntactically by a keyword instead of an argument. In the previous example, `receive:on:` is one such method, and `activate:during:` and `enqueue:in:` are two such messages. The syntactic support of receiverless methods and messages is natural since the first argument does not play a distinguished role in object interactions. Semantically there is no distinction between "receiverless" and "receiverful" methods. Therefore, these two appellations are not part of Ambience's parlance.

The syntactic symmetry of multimethods is orthogonal to the symmetry of the *multiple-dispatch semantics* used to select these methods at run time. Multiple dispatch is called *symmetric* if the rules for dynamic method lookup treat all dispatched arguments identically, and *asymmetric* if a linearisation or ordering is used — typically a lexicographic one, with earlier arguments having priority over later arguments in the selection between equally specific methods [3]. Ambience *multimethods* are symmetric, but the *multiple-dispatch semantics* is asymmetric, as in Slate and CLOS.

To continue with the example, suppose now that a particular phone `bobs-phone` delegates to the `mobile-phone` prototype, and that an incoming call `alices-call` delegates to the `phone-call` prototype; then the following message will trigger the execution of the `receive:on:` method defined previously:

```
receive: alices-call on: bobs-phone
```

If `alices-call` did not delegate (directly or indirectly) to `phone-call`, or if `bobs-phone` did not delegate (idem) to `mobile-phone`, then the method would not be applicable, and the message would not be understood.

To illustrate method overriding, consider a version of the method `receive:on:` that is specialised on smartphones rather than on regular mobile phones:

```
receive: call (phone-call) on: phone (smartphone)
[ show: phone ring-animation on: phone display.
  resend ]
```

This method will be invoked if the device receiving the phone call delegates to the `smartphone` prototype. The method displays an animation on the screen and invokes the overridden method behaviour by means of a `resend` call, analogous to `resend` in Self [18], to `call-next-method` in CLOS or to a `super` call in Java and Smalltalk. The `resend` message results in the invocation of the next most-specific method that is applicable for the actual arguments passed to the current method. Supposing that `smartphone` delegates to `mobile-phone`, the method invoked by the resend message will be the version of `receive:on:` specialised on regular mobile phones (shown previously), which will make the phone ring and handle the call.

3.2 Subjects for context-aware adaptation

Our approach to context-aware behaviour adaptation can be regarded as an instance of *context-oriented programming* [4] combined with the notion of *subjective objects* [17]. The main idea is that object behaviour, exhibited in response to a message send, does not only depend on the message arguments, but also on the message sender [12]. That is, the "point of view" of the caller affects behaviour selection.

The dependency on the caller's perspective is realised in Ambience by means of *subjective dispatch*, a mechanism originally found in the Slate programming language [16], which in turn draws inspiration from Smith and Ungar's notion of subjective objects. Technically, the point of view of the caller is reified as a plain object. Whereas related approaches call such object a *subject* or *layer*, we call it a *context*. The nature of this context object is explained next.

3.2.1 Context structuring

Context objects, as any normal object, can delegate part of their behaviour to other objects. These delegate objects can be seen as representing domain-specific *subcontexts* of the original context. Subcontext objects can delegate further to finer-grained subcontexts as needed. Figure 1 shows a hypothetical context configuration for ambient-oriented computing. Even though this sample graph is a tree, most often it will not be the case.[7] The `display`, `speakers` and `2d-input` (mouse) objects in the figure are *not* reifiers of the corresponding hardware parts. These context objects rather represent the presence or availability of such hardware resources in the current context. Similarly, `office` does not reify the physical room where the device is located: it represents the fact that the device is inside an office. In general, context objects *signal properties* of the context.

Note that Figure 1 shows only the behavioural part of the context, that is, the delegation links. That does not mean that the context is limited to storing delegation informa-

[6] Maintaining the resemblance to plain English, argument specialisers have the form of parenthetical expressions that clarify the kind of arguments the method is about.

[7] An example of a non-simplified context graph is given in Figure 2.

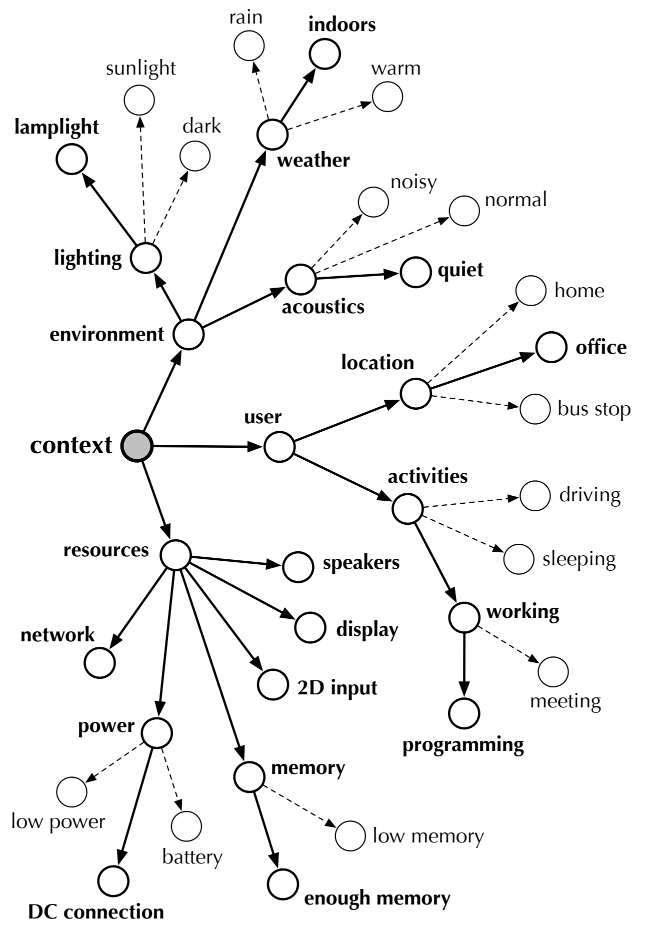

Figure 1. Context organisation example. The main context delegates to different domain-specific subcontext objects. Dotted arrows show alternative delegation possibilities.

tion exclusively. Context objects comprised in the graph can also contain plain slots with arbitrary contextual information needed by applications. Such slots account for the data-oriented part of the context. Having said that, our research focuses on the behavioural part of the context.

3.2.2 Influence of context on object behaviour

The object labelled `context` in Figure 1 serves as a handle to all subcontext objects. It constitutes the *current context* of the system, and is part of the state of the interpreter. In Ambience, the current context is passed implicitly as the first argument of every message.[8] Correspondingly, an extra formal argument is added implicitly to every method definition, using the current context as implicit argument specialiser. Methods are thus specialised on their context of definition. Therefore —following the multiple dispatch semantics explained in Section 3.1— a method is applicable only when its (sub)context of definition can be found in the current context graph, starting from the root context object. This implicit argument and the interplay with the multiple dispatch semantics constitute the core of subjective dispatch. Note that the Prototypes with Multiple Dispatch model needs no modifications in order to support subjective dispatch, other than the addition of an implicit context argument. The model thus keeps its original simplicity.

As an example of the subjective dispatch mechanism just described, consider again the method `receive:on:` introduced earlier in this section. The method could be defined to behave differently depending on whether the `acoustics` subcontext illustrated in Figure 1 delegates to a `quiet`, `normal` or `noisy` context. The first version, which avoids making noise, can be used in places such as libraries and situations such as meetings:

```
in-context: quiet do:
[ receive: call (phone-call) on: phone (mobile-phone)
  [ activate: phone vibrator during: 10 seconds.
    enqueue: call in: phone incoming-calls ] ]
```

The `in-context:do:` call switches the current context to `quiet` and evaluates the passed code block within that context. Since the code block contains the `receive:on:` method definition, the defined method will be specialised on the `quiet` context. The second version can be used in noisy places:

```
in-context: noisy do:
[ receive: call (phone-call) on: phone (mobile-phone)
  [ activate: phone lound-ringtone during: 20 seconds.
    activate: phone vibrator during: 20 seconds.
    enqueue: call in: phone incoming-calls ] ]
```

This version of the `receive:on:` method will be specialised on the `noisy` context. If the `acoustics` context happens to delegate to `quiet`, then the first version will be applicable. However, if the delegation is switched from `quiet` to `noisy`, then the second version of the method will come in force. As Salzman and Aldrich point out [16], "prototypes naturally support composition of subjects by delegation, allowing for a sort of dynamic scoping of methods by merely linking contexts together with dynamic extent."

Whenever a subcontext object is reachable from the current context object, the subcontext is said to be *active*. The dotted arrows in Figure 1 show possible alternative subcontexts that can be activated or deactivated in real-time according to changes detected in the environment. The manipulation of delegation links at run-time gives rise to what we call *dynamic context switching*.

3.3 Dynamic context switching

The context graph topology is an instantaneous representation of the current real-world context. For instance, Figure 1 shows a situation in which the user, at that precise moment, is programming at her office (indoors, with lamplight, in a quiet environment), with the usual peripherals (mouse, screen, speakers) and abundant resources (DC

[8] Acually, it is the current method activation record that is passed, but the activation record delegates directly to the current context object.

power, enough memory, large network bandwidth). Such context can change anytime. For example, if the user leaves the building, the delegation link going from `lighting` to `lamplight` will be switched to `sunlight`. Dynamic inheritance is thus exploited to adapt the context graph such that it reflects the current real-world context as timely and accurately as possible. More precisely, context changes in the domain system are reflected in the computational system by delegation slot changes. We call each of these changes a *context switch*. A seemingly simple or unitary action such as moving from one room to another can give rise to many context switches in the context graph, each one reflecting a change at a different domain-specific level, such as variations in illumination and expected room noise. Furthermore, these changes can take place in the middle of the execution of applications. Such issues related to our particular representation of context and the way that representation is managed at run time are the topic of the next section.

4. Context management

The foundations of our approach to dynamic context-aware behaviour adaptation have been laid in the previous section. We model the context as an object graph that guides behaviour selection. Managing this object graph correctly is vital for behaviour coherence. This section shows the techniques we have built on top of the subjective approach to context adaptation described so far. These techniques aim at easing context management and maintainging behaviour coherence.

Let us revisit the scenario from Section 2.1 to illustrate some practical issues in managing the context and our proposed solutions. In doing so we also provide a second more advanced example of the approach introduced in Section 3. Recall that the CityMaps application from the scenario is about showing maps to the user. To draw the maps on the screen, the application has the following methods:

```
in-context: city-maps do:
[ draw: map (map-section) on: display (canvas)
  [ ”proceed from the background to the foreground”
    draw: map background on: display.
    draw: map buildings on: display.
    draw: map streets on: display.
    draw: map highlights on: display.
    draw: map labels on: display ].

  draw: elements (collection) on: display (canvas)
  [ elements do: [ element | draw: element on: display ] ].

  draw: street (avenue) on: display (canvas)
  [ code to draw an avenue ].

  draw: street (highway) on: display (canvas)
  [ code to draw a highway ].

  . . . ]
```

Here the `city-maps` context represents a situation where the CityMaps application is available. The methods defined in this context provide the basic functionality of CityMaps.

4.1 Framework contexts

In the scenario, the availability of a GPS service in the environment renders the CityMaps application more navigational. We thus need extension code that accounts for situations where CityMaps and a GPS are both active simultaneously. The code should change the behaviour of CityMaps such that the current geographical location is taken into account when the map is displayed.

First of all, we need a GPS framework that allows applications to cooperate irrespective of the particular GPS service provider. We define a prototypical `gps` context that contains a `gps-locator` prototype and three unary methods understood by the locator:

```
in-context: gps do:
[ define: #gps-locator as: object clone

  (gps-locator) latitude
  [ 0 ”return Equator latitude” ].

  (gps-locator) longitude
  [ 0 ”return Greenwich meridian longitude” ].

  (gps-locator) coordinates
  [ latitude paired-with: longitude ].
```

These three method definitions illustrate the syntactic form of unary methods in Ambience. However, these methods are special in that they have anonymous arguments. When an argument is not used in the method body, its name can be omitted from the header. The `gps-locator` prototype is the argument specialiser of the three methods. The `latitude` and `longitude` methods return a default value,[9] since actual geographical data can be determined only by implementers of the framework. The `gps` context is analogous to the `display` context of Figure 1: it signals the presence of a hardware resource in the environment. If `gps` is active (i.e. if it is present in the current context graph), it means that the GPS service of some provider is currently available. The `gps-locator` prototype and the `longitude`, `latitude` and `coordinates` methods of the framework conform a *common vocabulary* for applications dealing with GPS. This framework illustrates the way requirement 1 of Section 2.2 can be addressed.

ACME, the vendor that develops GPS modules for smartphones, provides an instance of the framework that is specific to their particular hardware:

```
define: #acme-gps as: gps extension.
```

[9] As in Slate, the value returned by a method is that of the last evaluated expression. Hence the methods do not use the return operator (ˆ).

```
in-context: acme-gps do:
[ (gps-locator) latitude
  [ read latitude from hardware ].

  (gps-locator) longitude
  [ read longitude from hardware ] ]
```

The vendor-specific `acme-gps` context delegates to the more general `gps` framework context. The `latitude` and `longitude` methods are overridden, while the `coordinates` method is inherited by delegation. When ACME's hardware module is connected to the smartphone, its detection will trigger the activation of the `acme-gps` context. As a consequence of the delegation link, also the `gps` context will become active. Note that the generic `gps` context will never be activated on its own. Doing so would render the generic `latitude` and `longitude` methods applicable, but these methods do not provide real GPS information. Framework contexts are always delegates of some concrete context in the current context graph.

4.2 Context combinations

We have shown the independent code of CityMaps on the one hand and of a generic GPS framework and ACME's customisation of that framework on the other hand. Both parts have been conceived separately. Now we need glue code that prescribes their interaction. Such interaction is not specific to the CityMaps application alone (i.e. to the `city-maps` context), nor is it to GPS alone (i.e. to the `gps` or `acme-gps` contexts). A *combined context* is needed, in which to define the cooperation:

```
1  in-context: { city-maps, gps } do:
2  [ draw: map (map-section) on: display (canvas)
3    [ map center-on: gps-locator coordinates.
4      resend.
5      user coordinates: gps-locator coordinates.
6      draw: user avatar on: display ] ].
```

When the `in-context:do:` method receives the list of contexts { `city-maps, gps` }, it creates a new context object that delegates both to `city-maps` and to `gps`. The new context represents the combination of the two original contexts. The code block containing the definition of the `draw:on:` method will be evaluated in the newly combined context. Therefore, this version of `draw:on:` will be specific to that particular combination.

The `coordinates` message sent to `gps-locator` on line 3 reads the current geographical location from the hardware. Note that the `coordinates` method is implemented within `gps`, yet the method will *not* invoke the versions of `latitude` and `longitude` implemented in `gps`. The reason is that prototype-based delegation does not suffer from the "self problem" [14]. Regardless of the delegated methods that are invoked, the implicit context argument remains bound to the value that was passed initially: the current context root. The `acme-gps` context will be found before the less specific `gps` context in the current context graph. Therefore, ACME's version of `latitude` and `longitude` will be invoked, querying the hardware.

Once the map section has been relocated according to the GPS coordinates just read, the `resend` message on line 4 invokes the next most-specific `draw:on:` method, which implements the default map-drawing behaviour. The last two lines draw the graphical representation of the user at her current location. The net effect is that, when the combined { `city-maps, gps` } context is in effect, the location of the map will correspond to the current geographical location, and the user will be represented graphically at the centre and on top of streets, buildings, and other map elements. CityMaps will thus have become navigational, as was intended in the scenario.

4.2.1 Uniqueness of context combinations

At all times, there is at most one context object representing the combination of a given set of component subcontexts. For instance, the combination of { `city-maps, gps` } always results in the *same* combined context object that delegates to `city-maps` and to `gps`. If it were not the case, that is, if a new context object delegating to `city-maps` and to `gps` were created each time it were needed, then the methods that were specialised on the first instance of the context combination would not be visible (applicable) on the second or any new subsequent instances that would be created, despite the fact that they conceptually represent the same combination. Conceptually there is only one { `city-maps, gps` } combination, and computationally this must also be the case. Furthermore, the order in which contexts are combined is irrelevant.

In summary, there is a one-to-one correspondence between *sets* of component subcontexts $\{c_1, \ldots, c_n\}$ and the context object that represents the combination. On a practical level, this uniqueness property implies that created combination objects need to be stored by the context management system of Ambience, so that these same objects can be retrieved when required.

4.2.2 Implicit combination of contexts

The invocations of `in-context:do:` that pass a literal list such as { `city-maps, gps` } are not the only points at which contexts are combined. Actually, this kind of *explicit* combination —the only kind shown so far— is scarce. Most context combinations are performed implicitly by the system and on the fly, as environment changes are detected. To this end, the current-context object delegates to the combination of all contexts that have been switched on dynamically. Whenever a new context is switched on, it is combined with the active contexts that make up the current combination. For example, if the current combination is { `sunlight, city-maps, noisy` } (e.g. the user is using the CityMaps application on the street) and `acme-gps` is activated (e.g. the user plugs in the GPS hardware module), then the system will create a new combination { `sunlight, city-maps, noisy, acme-gps` }, and such combination will be activated.

4.2.3 Delegation among combined contexts

Combined contexts that are more specific than other existing combinations must delegate to those combinations. In the previous example, the { `sunlight, city-maps, noisy, acme-gps` } combination context should have a delegation link to the { `sunlight, city-maps, noisy` } combination context, since the former corresponds to a superset of the latter. The delegation link makes sense conceptually because supercombinations, as we call them, are more specific (contain more information about the context) than subcombinations. The least-specific combinations are those of only one subcontext. In these cases, a new object that represents the combination is not created; rather, the sole subcontext is taken as representation of the combination. For example, the combination of the set { `city-maps` } is the context `city-maps` itself. If two combinations are not comparable (neither is more specific), then no delegation link is established between them.

In determining the specificity of a combination (whether it is a supercombination or a subcombination of another one) it is not sufficient to examine its delegation links shallowly. Suppose that the current combination is { `sunlight, city-maps, noisy, acme-gps` }. That is, the CityMaps application with the GPS module is being used on the street. This combination should delegate to the combination of { `city-maps, gps` }; if it did not, the specialised `draw:on:` method (see the beginning of Section 4.2) would not take effect, and the user would not see any difference with respect to the plain CityMaps behaviour. The delegation link is thus needed. However, the set { `sunlight, city-maps, noisy, acme-gps` } is not a superset of { `city-maps, gps` }, since the former lacks element gps. However, by going one step further in the delegation hierarchy, we observe that `acme-gps` delegates to the missing `gps`. Thus, the first combination actually is more specific than the second. Hence, a delegation link should be established between them.

The description of our approach finishes by showing the way concurrent context updates can be managed.

4.3 Concurrent context manipulation

Context switches take place dynamically, as changes are detected in the surrounding environment. As a consequence, context switches occur concurrently, at the same time applications run on the device. Not all points in execution are safe to perform those context switches without affecting the behavioural consistency of the system. Consider again the `draw:on:` method specialised on the { `city-maps, gps` } combination from Section 4.2. Suppose that at a given moment line 4 is being executed, that is, the default map elements are being drawn by the original version of `draw:on:`. At this point the user decides to remove the GPS module from the smartphone.[10] The `acme-gps` context will thus be switched off, and as a consequence, also the delegate `gps` context will be deactivated. The smartphone is henceforth unaware of GPS. When control returns from the `resend` method and reaches line 5, the `coordinates` message will not be understood, since the `coordinates` method is specialised on the gps context that is no longer active. The problem, stated generally, is that a context has been switched off in the middle of the execution of a method that depends on that context to work correctly, invalidating the remainder of the computation of the method.

Brittle code that depends on unreliable resources such as network connections and removable peripherals could be surrounded by try/catch blocks. However, this solution would result in tangled, less readable code. Our solution is of another nature. Instead of passing directly from having a context to the absence of the context, we go through a series of intermediate contexts that allow us to gracefully degrade the system. Each stage can have specialised methods that deal with the situation and exhibit context-adapted behaviour. For our running example, we define one intermediate `degraded-acme-gps` context. The evolution of the context can be depicted as a chain of available services:

$$\ldots \rightarrow \texttt{acme-gps} \rightarrow \texttt{degraded-acme-gps} \rightarrow \varnothing$$

The disconnection of the GPS module will result in the deactivation of the `acme-gps` context, and in the activation of the `degraded-acme-gps` context. Note that the latter must also comply with the GPS framework contract (recall Section 4.1), so that the ongoing execution of methods that depend on the GPS service is not disrupted when `degraded-acme-gps` is switched on. We therefore have:

```
define: #degraded-acme-gps as: gps extension.

in-context: degraded-acme-gps do:
[ (gps-locator) latitude
  [ return extrapolated latitude ].

  (gps-locator) longitude
  [ return extrapolated longitude ] ]
```

After a predefined timeout of (for instance) 10 seconds, if the connection with the GPS service has not been regained, the `degraded-acme-gps` context can be deactivated. However, the previous methods are still insufficient for tackling our problem. When the system is running in degraded-GPS mode, the `draw:on:` method specialised on the { `city-maps, gps` } combination is still applicable, as explained next. The current combination contains `city-maps` and `degraded-acme-gps`, and the latter delegates to `gps`, so together these constitute a supercombi-

[10] In a similar scenario in which the GPS service were provided by a network peer, removing the hardware module would be analogous to losing the connection with the peer.

nation of { city-maps, gps }. In other words, there will be a delegation link that leads to the { city-maps, gps } combination. Since the draw:on: method is still applicable in degraded mode, it can be that the decision of removing degraded-acme-gps from the context is made at the wrong moment, precisely in the middle of the execution of draw:on:. This will raise exactly the same problem described previously for the deactivation of acme-gps. The solution is to define a degraded version of draw:on: that does not rely on GPS-specific functionality (i.e. that refrains from invoking methods from gps):

```
in-context: { city-maps, degraded-gps } do:
[ draw: map (map-section) on: display (canvas)
  [ resend.
    draw: disconnection-icon on: display notification-area ] ].
```

This method first invokes the plain map-drawing behaviour and then draws a notification icon telling the user about the loss of GPS signal. We use a new degraded-gps context instead of degraded-acme-gps, so that the degraded draw:on: method can be used with any vendor. The degraded-gps context extends the GPS framework.

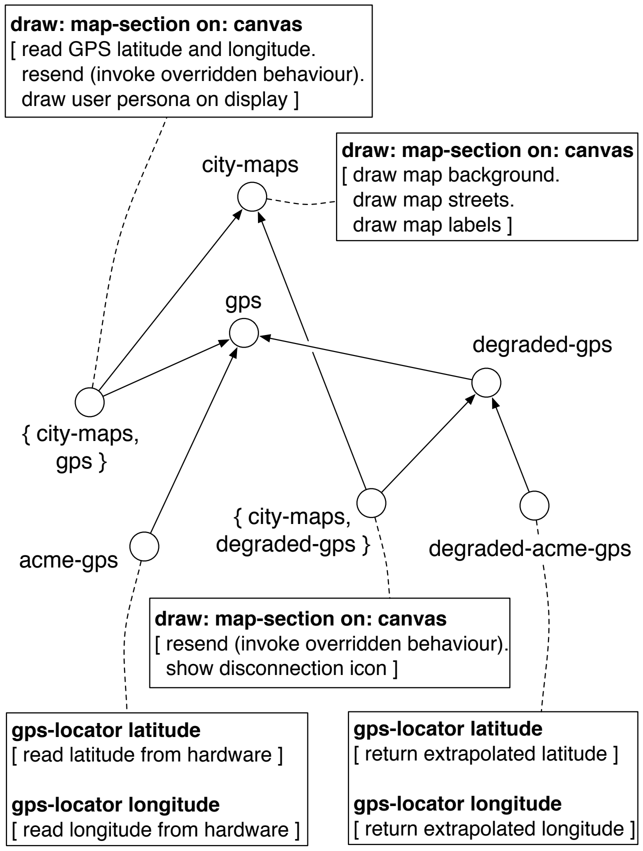
draw: map-section on: canvas
[read GPS latitude and longitude.
resend (invoke overridden behaviour).
draw user persona on display]
city-maps
draw: map-section on: canvas
[draw map background.
draw map streets.
draw map labels]
gps
degraded-gps
{ city-maps, gps }
{ city-maps, degraded-gps }
acme-gps
degraded-acme-gps
draw: map-section on: canvas
[resend (invoke overridden behaviour).
show disconnection icon]
gps-locator latitude
[read latitude from hardware]
gps-locator longitude
[read longitude from hardware]
gps-locator latitude
[return extrapolated latitude]
gps-locator longitude
[return extrapolated longitude]

Figure 2. Delegation relationships among context objects (arrows), and pseudo-code of methods specialised on those contexts (dotted lines).

A birds-eye view of the context configuration described so far is given in Figure 2. The net effect is that, instead of abruptly passing from the navigational CityMaps to the static CityMaps application, which would be disconcerting for the user, the degraded versions of the draw:on:, latitude and longitude methods will maintain the behavioural coherence of the application, ensuring a smooth context transition. After a given timeout, the degraded-acme-gps context will be switched off. The notification icon will no longer be shown and the map will be static again. The GPS service will have been completely —and gracefully— removed from the system.

5. Discussion

The computation model we have chosen is highly dynamic. It features dynamic typing, dynamic inheritance and dynamic dispatch (multimethods). The combination of these last two features and the implicit context parameter give rise to subjective behaviour, which adds yet one more degree of dynamicity: observed method behaviour depends on the changing domain context. Our first experiments suggest that such dynamic model —prototypes with subjective multimethods— eases the development of ambient-oriented applications.

We take full advantage of the particular qualities of the model for ambient-oriented programming. The concreteness and malleability of the context representation is due to the use of prototypes. Known properties of prototype-based programming [15] are used at their best: the objects in the context graph have *idiosyncratic behaviour* that is adapted to the particular context they represent (e.g. acme-gps); furthermore, these context objects can *delegate* their behaviour to more general, prototypical contexts (e.g. gps) without incurring the well-known "self problem" as explained in section 4.2; finally, *dynamic inheritance* is used constantly to adjust system behaviour in correspondence to context changes.

5.1 Meeting the requirements of applications

The model meets the requirements of ambient-oriented applications put forward in Section 2.2 as follows:

Shared vocabularies (Req. 1) Prototypical objects are a natural choice for constituting shared vocabularies that allow applications to interoperate. Framework contexts such as gps or noisy are particular cases of prototypical objects, that settle a common basis on which programmers can describe specialised behaviour that is reusable in specific (prototypical) situations.

Context representation (Req. 2b) The context representation we have chosen allows timely updates since these amount to changing the value of delegation slots in the context graph, which are very lightweight operations. Regarding accuracy, we noted in Section 3.2.1 that the context can be divided into finer-grained subcontexts to any desired level of detail. Furthermore, we noted that the context graph is not limited to holding behaviour-oriented information; relevant context information that is not behavioural can also be maintained in the form of plain data slots for each domain-specific subcontext object.

Context effect (Req. 2c) The effect of the context representation on object behaviour could not be more direct and timely: changes in the context graph reflect immediately on the methods that are chosen for any subsequent message dispatch. Whereas the timeliness of context effects is guaranteed by our approach, the accuracy depends on the particular multiple-dispatch semantics of the language. As discussed next, the particular method linearisation or disambiguation technique (in asymmetric or symmetric dispatch respectively) can affect directly the behavioural coherence of the system.

5.2 Multiple dispatch and ambiguities

In a language with symmetric multiple-dispatch, the programmer must disambiguate methods explicitly. Symmetric dispatch is seen as more intuitive and less error-prone, since possible ambiguities are reported, rather than being silently resolved in potentially unexpected ways by the runtime system [3]. However, the high interaction level that is characteristic of ambient computing makes it very difficult —if not impossible— to foresee the ambiguities that could happen in practise, since methods can be exchanged in unanticipated ways between devices. Furthermore, the simple modification of a delegation link —a very common action in our approach— can render a set of methods ambiguous, if the newly inherited methods have the same selector than methods that were already visible with compatible specialisers. Clearly, compile-time detection of ambiguities as implemented in Cecil [2] is impossible in ambient-oriented languages. Even so, symmetric dispatch needs not be ruled out: it rather needs to resort to domain-specific disambiguators that are as transparent to the user as possible.

Asymmetric dispatch, on the other hand, never results in ambiguities because the set of applicable methods is ordered by some means, but it can lead to selection of behaviour that is inappropriate for a given situation [16]. The decision of which method is most appropriate for a given situation cannot always be taken automatically. We still need to carry out a full assessment of symmetric and asymmetric multiple-dispatch techniques for mobile systems. Hence, the choice of asymmetric dispatch in Ambience is not final. In Section 7 we suggest a line of future work that could provide us with an answer.

6. Related work

Context-oriented programming and subjective multimethods have been investigated to a relatively limited extent so far. However, the emergence of a few new object-oriented languages that bear subjective-like features (at least three: Slate, ContextL and Ambience) indicates that researchers are turning their attention to subjectivity again.

6.1 Slate

Ambience is inspired by the Slate programming language and its underlying Prototypes with Multiple Dispatch Model (PMD) [16]. The object system of Ambience uses PMD as its kernel: classless objects, multimethods, and the roles that bind them together. Slate adds many features on top of PMD that Ambience does not incorporate such as optional typing and optional method arguments. Starting from the basic PMD model, Ambience has evolved in its own direction, driven by the needs of ambient-oriented applications. The main driving force has been the one described in this paper, namely subjective multimethods for dynamic behaviour adaptation. Even though Slate does include a subjective dispatch mechanism similar to ours, and the authors of Slate are well aware of the potential of subjective dispatch, the mechanism has been relegated to a second plane. Currently, subjective dispatch is a disabled feature in the implementation of Slate, and we know of only one example of its application [16]. Ambience incorporates subjective dispatch and boosts it to a point that it becomes as fundamental to the programming model as prototypes and multimethods.

6.2 ContextL

ContextL —an extension of CLOS [1]— not only shares the similar goal of having behaviour depend on the context, but also a similar approach. ContextL provides means to associate partial class and method definitions with layers —the analogues of contexts in Ambience— and to activate and deactivate such layers in the control flow of a running program [4]. In ContextL the representation of the currently active layers is analogous to the current-combination object of Ambience. This representation is passed implicitly among methods, and serves as a parameter of an auxiliary multimethod that provides the layering semantics of ContextL. This basic mechanism bears close resemblance to subjective dispatch in Ambience. However, the layer argument has the least precedence in the argument precedence order of the auxiliary multimethod,[11] whereas in Ambience the context argument has priority over the other arguments.

ContextL offers a `with-active-layers` construct for activation of layers with dynamic scope that is similar to Ambience's `in-context:do:`. An important difference is that ContextL is enterely based on this construct for layer activation —that is, activation and deactivation is tied up with dynamic scoping. The consequence is that context changes are necessarily visible in the program text as occurrences of `with-active-layers` and `with-inactive-layers`. In Ambience, context-switching code is seldom seen inside a method implementation (calls to `in-context:do:` can be used, but the idea is that application code should not be cluttered this way). The most frequent —practically exclusive—

[11] This is the default behaviour, but CLOS provides means to change the argument precedence order, and by extension so does ContextL.

use of explicit `in-context:do:` calls is to define context-specific methods, as has been seen in the examples.

Next to the `in-context:do:` method for dynamic scoping, there are two methods `switch-context-on:` and `switch-context-off:` in Ambience that can perform activation and deactivation of contexts irrespective of dynamic scoping, in a more imperative fashion that contrasts with the functional flavour of `in-context:do:`.[12] Furthermore, these methods can be invoked from another (monitoring) thread, yet affect the thread that runs the base code. In ContextL, on the contrary, one thread cannot change the active layers of another thread.[13] Whereas thread locality ensures non-interference with other threads, such interference is sometimes desired. For example, if the user is on the street —where it is `noisy` and there is `sunlight`— and then enters a building —where it is `quiet` and `dark`— which thread should be notified of the context changes? In ContextL, devices have as many current contexts (layer activation representations) as running threads. Conceptually however, the (real-world) context is only one. One solution in ContextL would be to replicate the context changes among the layer combinations of the different threads. Even then, these changes would take effect at the top-level only. Methods that would be already running would not see the change (which goes against the coherence requirements explained in Section 2.2). In Ambience, there is only one context graph that is shared by all threads, and context switches are performed concurrently with the base running programs. Application behaviour is adapted on the fly. However, concurrent context switching, whereas needed, brings its own problems as discussed in Section 4.3. Furthermore there might be situations in which it is desirable to affect the context of one running application (thread), and let the others be oblivious of the change. We believe that the approaches of ContextL and of Ambience can be complementary, and that some combined approach may be worth investigating.

7. Future work

The first assessment of Ambience presented in this paper leads us to believe that the programming model is viable and well suited to the task of ambient-oriented application programming. However, much work remains.

An important area in which we need to carry further research is the treatment of ambiguities in messages, risen by multiple inheritance and multiple dispatch. As discussed in section 5.2, automatic linearisation mechanisms such as lexicographic orderings are not always desirable, and user-defined mechanisms on the other hand require good engineering insight. We are considering the adoption of *context conditions* and *composition rules* [7], which use a logic reasoning engine to tackle the problems of interactions that arise in the composition of context-dependent adaptations.

Although research on Ambience concentrates on behaviour adaptability and on context awareness, ambient-oriented applications pose technological challenges at many different levels, including the problems of dealing with connection volatility, peer-to-peer communication and decentralised coordination of activities. Issues like these have been addressed by the AmbientTalk language [6], but not yet by Ambience. While deciding on the feature set of Ambience, we have been mindful of AmbientTalk's provisions for ambient-oriented programming, in the hope that the integration of those provisions will not be hampered by incompatible features of Ambience. For example, we would like to extend the model of Ambience with reified mailboxes [5] (queues of incoming, outgoing, received and sent messages). However, AmbientTalk and all other existing Actor-based languages rely strongly on asymmetric, singly-dispatched methods where there is a distinguished receiver object to which message queues are associated. The assignment of messages to the incoming and outgoing queues of objects is no longer clear in a language with symmetric multimethods, since all message participants are on a same footing. We believe that the freedom of choosing the participant that will process a given message can be exploited to implement (for example) load-balancing mechanisms, where methods are executed by participants residing in the more resourceful machines.

Our main short-term goal is to validate the approach more thoroughly and gain further experience by writing and testing additional ambient-oriented scenarios. At the same time, we will be able to distill a first programming methodology that will guide programmers in designing the structure and behaviour of contexts, and more generally, in programming with dynamicity and behaviour variability in mind.

8. Conclusion

The advent of mobile technology has raised expectations on the kinds of applications that can be built for the new platforms. The new applications are expected to be smart with respect to the changing environment that surrounds them and to the user needs. It follows that the behaviour of this kind of applications should be adaptable in a context-aware fashion. A first contribution of this paper is to present a highly dynamic object-based computation model —based on prototypes with subjective multimethods— that helps to achieve such dynamic context-aware adaptability of object behaviour. The solution is validated by implementing a real–life-like scenario in Ambience, a new language we have developed for this purpose.

The second contribution of this paper is to propose a concrete, malleable representation of context that can be updated efficiently, as context changes are detected in the en-

[12] The methods `switch-context-on:` and `switch-context-off:` can be regarded as performing destructive assignments in the context graph, whereas `in-context:do:` follows a non-destructive, stack-like discipline.

[13] This design decision is an inherited one: Common Lisp implementations follow the practise of creating dynamic bindings on a per-thread basis.

vironment. Furthermore, the changes in the representation have an immediate effect on object behaviour. As no inherent latencies are introduced, environmental changes reflect timely on observed application behaviour. Also, the representation allows the modularisation of context in domain-specific subcontexts, to any desired level of granularity. Behaviour can thus be tweaked and adapted accurately with respect to the application domain.

We have proposed a context combination mechanism that helps aggregating all the domain-specific properties of the changing environment on the fly, as applications execute. In coming up with this mechanism we have found that our solution permits a form of graceful service degradation, by supporting smooth progressions of intermediate contexts, rather than have radical or abrupt context changes take place.

Further research is needed to devise a way to deal with ambiguities, when more than one method is applicable for a given message in the current context. These ambiguities are not necessarily programming errors: it can be that more than one behaviour is correct for a given situation, even though some behaviours might be more adequate than others. Furthermore, we need to integrate language abstractions for concurrency and distribution that have been proposed by other approaches, in such a way that they are compatible with the characteristics of our approach, particularly with symmetric multimethods.

9. Acknowledgments

This research has been supported by the Fund for Research Training in Industry and Agriculture (FRIA, Belgium) and by the Interuniversity Attraction Poles Programme of the Belgian Science Policy (Belgian State).

References

[1] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kicsales, and D. Moon. Common lisp object system specification. *Lisp and Symbolic Computation*, 1(3/4):245–394, 1989.

[2] C. Chambers. Object-oriented multi-methods in cecil. In O. L. Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP)*, volume 615, pages 33–56. Springer-Verlag, 1992.

[3] C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers. MultiJava: Design rationale, compiler implementation, and applications. *Transactions on Programming Languages and Systems (TOPLAS)*, 28(3), May 2006.

[4] P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming: an overview of contextl. In *Dynamic Languages Symposium (DLS)*, pages 1–10. ACM Press, Oct. 2005. Co-located with OOPSLA'05.

[5] J. Dedecker and W. V. Belle. Actors for mobile ad-hoc networks. In L. Yang, M. Guo, J. Gao, and N. Jha, editors, *Embedded and Ubiquitous Computing*, volume LNCS 3207 of *Lecture Notes in Computer Science*, pages 482–494. Springer-Verlag, Aug. 2004.

[6] J. Dedecker, T. V. Cutsem, S. Mostinckx, T. D'Hondt, and W. D. Meuter. Ambient-oriented programming. In *Companion to the annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 31–40. ACM Press, 2005.

[7] B. Desmet, J. Vallejos, P. Costanza, and R. Hirschfeld. Layered design approach for context-aware systems. In *Proceedings of 1st International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2007)*, pages 157–165. Technical Report at Irish Software Engineering Research Centre (Lero), Jan. 2007.

[8] S. Dobson and P. Nixon. More principled design of pervasive computing systems. In *Engineering for Human-Computer Interaction and Design*, volume 3425 of *Lecture Notes in Computer Science*, pages 292–305. Springer-Verlag, 2005.

[9] B. Foote, R. E. Johnson, and J. Noble. Efficient multimethods in a single dispatch language. In A. P. Black, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, LNCS 3586, pages 337–361. Springer-Verlag, 2005.

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

[11] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[12] W. H. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 411–428. ACM Press, 1993.

[13] ISTAG. Ambient intelligence: from vision to reality. Technical report, Information Society Technologies Advisory Group of the European Commission, 2003. Available at http://www.cordis.lu/ist/istag-reports.htm.

[14] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In N. Meyrowitz, editor, *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 21, pages 214–223. ACM Press, 1986.

[15] J. Noble, A. Taivalsaari, and I. Moore, editors. *Prototype-Based Programming: Concepts, Languages and Applications*. Springer-Verlag, 1999.

[16] L. Salzman and J. Aldrich. Prototypes with multiple dispatch: An expressive and dynamic object model. In A. P. Black, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, LNCS 3586, pages 312–336. Springer-Verlag, 2005.

[17] R. B. Smith and D. Ungar. A simple and unifying approach to subjective objects. *Theory and Practice of Object Systems (TAPOS)*, 2(3):161–178, 1996.

[18] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 227–242. ACM Press, 1987.