Prototypes with Multimethods for Context-Awareness

Sebastián González[1]*, Kim Mens[1], and Stijn Mostinckx[2]**

[1]Département d'Ingeniérie Informatique
Université catholique de Louvain, Belgium
{s.gonzalez,kim.mens}@uclouvain.be

[2]Programming Technology Lab
Vrije Universiteit Brussel, Belgium
smostinc@vub.ac.be

Abstract The paper proposes a concrete notion of context as an object graph which allows the representation of different sub-domains of contextual information. The behaviour of the system is implicitly influenced by such context. System behaviour can be adapted dynamically by manipulating the context graph, and also by introducing multimethods specialised on desired sub-contexts. The proposed approach aligns naturally with the underlying computation model of prototypes with multiple dispatch.

1 Introduction

The introduction of mobile devices equipped with wireless network provisions, allows for present-day mobile applications to become aware of their environment and to interact with it. Unfortunately, the incorporation of context information into running mobile applications is currently often achieved using *ad hoc* mechanisms. To allow for an application to behave differently in a given context, this context-specific behaviour is typically hard-wired in the application under the form of if-statements scattered in method bodies or by using design patterns [8] (*e.g.* Visitor, State, Strategy) and best-practice patterns [2] (e.g. Double Dispatch). As an alternative solution, in this paper we explore the Prototypes with Multiple Dispatch (PMD) object model [13] under the light of context-aware mobile applications. Our proposal provides a structured mechanism to deal with contextual information in a flexible and fine-grained manner.

Context-aware mobile applications rely on a context architecture that represents the input from sensors (and possibly other applications) in a way that is accessible to the application. The architecture used in this paper is akin to that of the Context Toolkit [6], using objects to aggregate the context derived from different (interpretations of) sensor data. The chief difference with our approach lies in the way the context architecture will be employed by the applications

* S. González is funded by the Fonds pour la Formation à la Recherche dans l'Industrie et dans l'Agriculture (FRIA, Belgium).

** S. Mostinckx is funded by the Institute for the Promotion of Innovation through Science and Technology (IWT, Belgium).

relying on it. To avoid hard-wiring context-related behaviour inside the application, the aggregated context directly influences the dispatch of methods. In other words, the programming model directly supports *Context-Oriented Programming* [4].

The structure of the paper is as follows. Section 2 presents the PMD model, the basis of our approach to context-awareness. Section 3 suggests the realisation of the application context as a simple object which affects system behaviour. Section 4 is about the aggregation of different kinds of context information into a single context object, reusing basic PMD mechanisms. Sections 5 and 6 show how the proposed model helps solving the problem of dynamic application adaptation to context. Section 7 reports on the current status and future plans for our work, section 8 reports on related work, and section 9 concludes the paper.

2 The PMD Model

The PMD model relies on a *prototype-based* object model, inspired by the programming language Self [17,12]. In prototype-based programming, objects are entirely self-sufficient such that they can properly function without requiring a class definition. Hence, each object contains its own variable and method slots. New objects are created by cloning existing *prototypes*, objects that act as representative examples of domain entities. However, prototypes do not have a special status in the language, as any object can be cloned and therefore serve as a prototype.

Objects in a prototype-based language can extend other objects by *delegating* to them [11]. The delegating object reuses the methods and variables of the delegate object. The relation between delegator and delegate is established using special *delegation slots*. Since delegation slots may change at run-time, this extension scheme is called *dynamic inheritance*, in contrast with standard inheritance in class-based languages, where inheritance relationships are defined statically.

The prototype-based object model of PMD is complemented with *multimethods* to avoid hard-wired dispatch code. Multimethods specify the kind of arguments for which they are designed to work, by means of *argument specialisers*. An argument specialiser is simply an object, usually a prototype. The following Smalltalk-inspired example shows a print:on: multimethod whose value and stream arguments are specialised on the string and output-stream prototypes respectively:

```
print: value (string) on: stream (output-stream)
[ . . . print the string . . . ]
```

The method body (between brackets) has been omitted for clarity.

Given a message, a multimethod is applicable if each argument specialiser is in the delegation graph starting at the corresponding actual argument of the message, as illustrated in figure 1. In the example, the print:on: method will be invoked only if the passed value argument delegates to string, or it is the

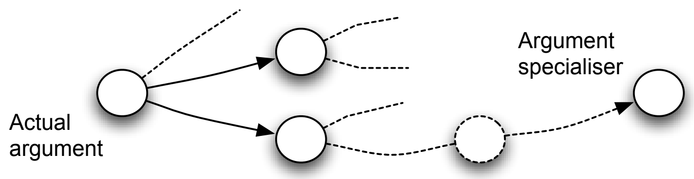

Figure 1. Directed (possibly cyclic) delegation graph showing the applicability of an argument. The dashed paths and nodes represent arbitrarily complex intermediate subgraphs. The actual argument is applicable if there is a path leading to the argument specialiser.

string object itself. An analogous rule applies to stream and output-stream. Since every method argument is treated in the same way, multimethods in PMD are said to be *symmetric*, in contrast with *asymmetric* multimethods where there is a distinguished receiver to which the method belongs [7]. As a consequence of symmetry, PMD does not feature a special self or this keyword.

A multimethod can be overloaded by using different combinations of argument specialisers. Upon a message send, the delegation graph distance between actual arguments and found argument specialisers is minimised so as to determine the most-specific applicable implementation. A left-to-right precedence rule is used where necessary for disambiguation, but further details are irrelevant to this paper. Again as a consequence of symmetry, the dynamic value of all arguments, rather than only a receiver, is used for method selection. Patterns such as Visitor [8] and Double Dispatch [10] are unnecessary in the PMD model [3].

The declarative power of multimethods stems from dynamic overloading. The programmer can add special cases simply by adding new multimethods specialised on the right set of arguments, without modification of previously existing code. Multimethods provide a flexible and more declarative mechanism to describe the interaction of different kinds of objects.

3 Context as an Implicit Behaviour Parameter

Clearly, if a system is to be called context-aware, the current context should be able to affect system behaviour. The context *parametrises* the behaviour of the system. Hence, it is natural to devise the context as an object which is passed as an extra argument of messages, and correspondingly, which is used as argument specialiser in method definitions. This object contains all the information pertaining to the current real-world context, and such information can have an effect on the behaviour of the system as described below.

Given that the context is used ubiquitously, it is better to pass it as an implicit parameter, rather than forcing the user to clutter the source code by passing the context explicitly. Note that the implicit nature of the context has nothing to do with the semantics of the model, it is just a convenience. Since the context is an implicit method argument, there is no easy way (syntactically) to explicitly specialise a method on a particular context. Rather, the current context in which a method definition is found is used as an argument specialiser of the implicit context argument. As an example, consider a text-user-interface context, which represents the situation in which the system interacts with the user by means of text:[1]

```
define: #text-user-interface as: object clone.
```

Observe that text-user-interface is a plain object, created by cloning the prototypical object.[2] Methods whose behaviour is specific to text-based interfaces can now be defined within this context. In particular, an inform: method, whose purpose is to give the user a message, can be implemented within text-user-interface:

```
within: text-user-interface do:
[ inform: message
  [ print: message on: standard-output.
    print: newline on: standard-output ] ]
```

Syntactically, the method signature "inform: message" followed by the code block between square brackets [...] represents a method definition. The inform: method so defined is specialised on the text-user-interface context. The method will print the given message argument on the console, followed by a newline. Only within the particular text-user-interface context will this implementation of the inform: method be applicable. If the following code is executed:

```
inform: 'ready'.
```

the text "ready" will be printed on the display of any device with a text-based display. In a device with a graphical user interface, inform: could be implemented in a graphical-user-interface context, and open a message box to show the message. This way, an application can achieve its semantic goal (to tell the user a message) on both devices without modification. The final application behaviour is not fixed and can adapt to the environment –the context– in which it runs.

4 Context Aggregation

In reality, the context is a complex entity, comprising information of various kinds. In our approach, the context –as any normal object– can delegate part of its behaviour to other objects. These objects can be seen as *sub-contexts*, entities which represent subdomains of larger context domains. Sub-contexts can delegate in their turn to further refined sub-contexts, up to any desired level

[1] The syntax we use is similar to Smalltalk's. Statements read almost like sentences in English.

[2] In prototype-based programming, objects are created by cloning existing objects. This mechanism substitutes class instantiation, given that classes are not used in this paradigm.

of granularity. Hence, a given context object effectively aggregates all the sub-contexts specific to the current situation. The main context, which ultimately aggregates *all* context information, can be conveniently passed as one simple argument in method invocations.

We illustrate context aggregation by continuing the example of section 3. Suppose an application runs in a mobile device with a text user interface and it is held by an English-speaking user. This can be implemented by aggregating two delegates into the current (main) context:

```
current-context define-delegate: #user-interface as: text-user-interface.
current-context define-delegate: #localisation as: english.
```

The define-delegate:as: message is analogous to the message define:as: used above, excepting that the created slot is a delegation slot rather than plain slot. The slot named user-interface references the sub-context text-user-interface already introduced. The slot named localisation references the english sub-context. This sub-context represents an English-speaking environment. Suppose we want to localise the inform: method so that it prints monetary values according to the English language standards. The print:on: method introduced above (see the implementation of inform:) can be specialised on monetary values within the english context, as follows:

```
within: english do:
[ print: amount (money-amount) on: stream (output-stream)
  [ print: '$' on: standard-output.
    print: amount value on: standard-output ] ].
```

The money-amount and output-stream objects specialise the amount and stream arguments. If the following code is executed:

```
inform: 5 dollars.
```

"$5" is printed on the text-based interface.

The same print:on: method can be defined for a Spanish-speaking context:

```
within: spanish do:
[ print: amount (money-amount) on: stream (output-stream)
  [ print: amount value on: stream.
    print: ' dólares' on: stream ] ].
```

Upon executing a change of localisation to Spanish:

```
current-context localisation: spanish.
```

the same application code (inform: 5 dollars.) will print "5 dólares" instead of "$5".

5 Dynamic Fine-grained Adaptation to Context

Adaptation to context occurs by altering the sub-contexts of the current (main) context, i.e., by changing delegation links as illustrated in section 4 with the localisation slot of current-context. Such context changes should occur dynamically, in order to reflect real world changes. Furthermore, adaptation to context must take place as transparently as possible to the user. Obviously, the user must not be queried by the system to find whether e.g. she is currently talking on the phone or reading news. Rather, this information is to be supplied by agents which monitor the environment and the user. The agents should dynamically modify context delegation links according to the information they sense or infer. Some delegation links of context objects might vary constantly. For instance, a user-activity context link will change whenever the user switches activities (reading, talking, walking, entering a given room, etc.). Other delegation links may remain relatively stable, such as the localisation link introduced above. Changes in the context's delegation graph reflect immediately on the behaviour of the application. This is our approach to realising dynamic application adaptation to context. We limit our attention to context changes which are easy to detect. The way sophisticated monitor agents should detect environment changes by analysing sensor data and affect delegation links accordingly is out of the scope of our work.

Every message send is a potential hook in which behaviour can be adapted, since any method can be overridden in particular contexts. Such high level of granularity in the hook points of a system benefits unanticipated, non-intrusive application adaptation (unanticipated because hook methods need not be planned in advance, non-intrusive because application code does not need to be modified, as illustrated with the inform: example in section 4).

6 Support for Explicit and Implicit Context Handling

There are two possible ways the context can affect system behaviour. The first (preferred) way to affect system behaviour is *implicit* and more declarative, and has already been described above. By means of multiple dispatch and implicit context arguments, the system can "take decisions" and change behaviour without hard-coded control flow, according to the current context.

Despite our preference for implicit context handling, we would like to note that *explicit* context handling by imperative means (i.e. hard-coded control flow) is supported as well. Since (sub)contexts are normal objects, they can contain not only delegation slots, but also plain slots with information about the current environment. This information can be queried programmatically and appropriate action can be taken, hence supporting explicit context handling. The context object graph can contain at the same time the information used for *both* implicit and explicit context handling.

7 Current Status and Future Work

The computation model partly described in this paper has been implemented in the Ambience virtual machine. The Ambiance VM consists of a core written in Common Lisp on top of which we laid a Smalltalk-like syntactic layer (shown in the paper). Even though the implementation is currently operational, a good deal of development and research work is ahead of us.

Firstly, we are adding support for distribution. To the extent of our knowledge, the possibilities of multiple dispatch in a distributed setting –let aside a mobile computing setting– are largely unexplored. Although it is possible to distribute the delegation relationship across different machines [16,5], we need to work on the way the multiple dispatch algorithm behaves when a particular delegate object is unavailable (e.g. because of the temporal disconnection of a peer device).

Secondly, we will be improving the model from a language security standpoint. While the basic principle of object encapsulation is supported (by allowing a slot to be visible only from certain contexts), there are some rough edges concerning security. Our approach to context adaptation in principle allows untrusted clients to override a method implementation with their own, by (re)defining a method in an appropriate sub-context. Code executing in super-contexts will start using the overridden version. A second problem is that calling non-trusted code (e.g. library code received from another device) from a privileged context will confer the caller's privileges to the untrusted code, which should have rather been run with more restricted privileges. To solve these security flaws, we are thinking of adapting composable encapsulation policies to our model [14], so that different clients obtain object references which grant different capabilities.

Thirdly, we have explored concurrency in our model to a limited extent. We implemented thread-based concurrency, where multimethods are always executed by the same thread that invoked them, unless a new thread is explicitly created. An alternative –perhaps more appealing– solution that aligns well with the notion of self-sufficient objects is to employ an actor-based concurrency model [1]. Actors are active objects (equipped with their own thread). However, since symmetrical multimethods do not distinguish a special receiver, all actors involved in a message interaction are equally eligible. We see in this choice an opportunity to explore the possibilities of the mobile computing world. The selection of a particular active object for message execution could be based on load-balancing criteria, on the reliability of the connection to remote devices, etc.

8 Related Work

The idea of using the point of view of the caller (in our case the current context) for affecting system behaviour is inspired on the language Us [15]. As the authors of Us state, there is no single "true" state and behaviour for an object:

rather, the state and behaviour of an object depends on a “perspective” which is reified as an object. US demonstrated the usefulness of subjectivity in common scenarios. Our proposal is similar to Us, excepting that Us proposed the receiver and one extra perspective or “point of view” to be participants in the lookup algorithm, whereas we use full (symmetric) multiple-dispatch, where all arguments are dispatch participants.

The closest existing approach to ours is that of ContextL, the implementation of Context-Oriented Programming in Common Lisp [4]. In ContextL, 1) the influence of context is described declaratively, 2) context information is passed implicitly through the dynamic invocation chain, 3) layer activations (i.e. `within:do:` messages) can be nested, and 4) context objects are aligned with threads, such that different contexts may be activated in different threads.

Even though there are many similarities with ContextL, some important differences take the two approaches apart. Firstly, ContextL is class-based. The problem of dynamic instance reclassification in class-based languages is not trivial. The State pattern [8] is a symptom of this problem, when systems need to behave as if they changed classes dynamically. In contrast with ContextL, our model is prototype-based, which allows us to dynamically manipulate the inheritance (delegation) graph of contexts naturally, avoiding the problems introduced by classification [12].

Another important difference with ContextL is that in our approach the context is reified as an ordinary object, which can be handed out to other threads. Therefore one thread may add or change a sub-context of the context of another thread. This can be the case of the monitor agents suggested in section 5.

9 Conclusion

The advantages of multiple dispatch are well known [7], and to a lesser extent, the advantages of multiple dispatch in combination with prototype-based programming [13,3]. Nevertheless, the use of prototypes and multiple-dispatch for context-oriented programming is largely unexplored. In this paper we have shown how a context-influenced dispatch mechanism can be used for fine-grained, dynamic adaptation of mobile applications to context. The notion of context we propose allows the aggregation of all necessary context information into one single entity which is automatically passed around and implicitly influences the behaviour of the system. Our solution reuses existing language features, avoiding the addition of complexity to the computation model. Concurrency, distribution and language-level security are to be further explored in the light of the proposed approach.

References

1. G. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

2. K. Beck. *Smalltalk: best practice patterns.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.
3. 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, Berlin, Heidelberg, New York, Tokyo, 1992. Springer-Verlag.
4. P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming. In *Dynamic Languages Symposium at OOPSLA '05: Companion of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* ACM Press, 2005.
5. T. V. Cutsem, S. Mostinckx, W. D. Meuter, J. Dedecker, and T. D'Hondt. Distributed proxies as delegation-based descendants. Technical Report VUB-PROG-TR-05-07, Vrije Universiteit Brussel, 2005.
6. A. K. Dey, D. Salber, and G. D. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, 16, 2001.
7. B. Foote, R. E. Johnson, and J. Noble. Efficient multimethods in a single dispatch language. In A. P. Black, editor, *ECOOP '05: Proceedings of the European Conference on Object-Oriented Programming)*, LNCS 3586, pages 337–361, Berlin Heidelberg, 2005. Springer-Verlag.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
9. W. H. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *OOPSLA '93: Proceedings of the 8th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 411–428, New York, NY, USA, 1993. ACM Press.
10. D. H. H. Ingalls. A simple technique for handling multiple polymorphism. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 347–349, New York, NY, USA, 1986. ACM Press.
11. H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 214–223. ACM Press, 1986.
12. J. Noble, A. Taivalsaari, and I. Moore, editors. *Prototype-Based Programming: Concepts, Languages and Applications.* Springer-Verlag, 1999.
13. L. Salzman and J. Aldrich. Prototypes with multiple dispatch: An expressive and dynamic object model. In A. P. Black, editor, *ECOOP '05: Proceedings of the European Conference on Object-Oriented Programming)*, LNCS 3586, pages 312–336, Berlin Heidelberg, 2005. Springer-Verlag.
14. N. Schärli, S. Ducasse, O. Nierstrasz, and R. Wuyts. Composable encapsulation policies. In *Proceedings ECOOP 2004 (European Conference on Object-Oriented Programming)*, LNCS 3086, pages 26–50. Springer Verlag, June 2004.
15. 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.
16. R. Tolksdorf and K. Knubben. dself - a distributed self. KIT-Report 144, TU Berlin, 2001.
17. D. Ungar and R. B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 227–242. ACM Press, 1987.