2nd Workshop on Object-Oriented Language Engineering for the Post-Java Era: Back to Dynamicity*

Sebastián González[1], Wolfgang De Meuter[2], Pascal Costanza[3], Stéphane Ducasse[4], Richard Gabriel[5], and Theo D'Hondt[2]

[1] Département d'Ingénierie Informatique, Université catholique de Louvain – Belgium
[2] Programming Technology Lab, Vrije Universiteit Brussel – Belgium
[3] Institute of Computer Science III, Universität Bonn – Germany
[4] Software Composition Group, Universität Bern – Switzerland
[5] Sun Microsystems – USA

Abstract. This report covers the activities of the 2nd workshop on "Object-Oriented Language Engineering for the Post-Java Era". We describe the motivation that led to the organisation of a second edition of the workshop. Relevant organisational aspects are mentioned. The main part of the report consists of a summary of Dave Thomas's invited talk, and a recount of the presentations by the authors of position papers. Comments given along the way by the participants are included. Finally, some pointers to related work and events are given.

1 Introduction

As stated in the workshop's first edition, the advent of Java has always been perceived as a major breakthrough in the realm of object-oriented languages. And to some extent it was: it turned academic features like interfaces, garbage-collection and meta-programming into technologies generally accepted by industry. Nevertheless Java also acted as a brake especially to academic language design research. Whereas pre-Java ECOOP's and OOPSLA's traditionally featured several tracks with a plethora of research results in language design, more recent versions of these conferences show far less of these. Those results that do make it to the proceedings very often are formulated as extensions of Java. Hence, they necessarily follow the Java doctrine: statically typed, single-inheritance, class-based languages with interfaces and exception handling.

Recent academic developments seem to indicate that a new generation of application domains is emerging for whose development the languages adhering to this doctrine will probably no longer be sufficient. These application domains

* The title of this report should be referenced as "Report from the ECOOP 2004 2nd Workshop on Object-Oriented Language Engineering for the Post-Java Era: Back to Dynamicity".

J. Malenfant and B.M. Østvold (Eds.): ECOOP 2004, LNCS 3344, pp. 49–61, 2004.

have recently been grouped together under the name of *Ambient Intelligence* (AmI). The visionary idea of AmI is that in the future, everybody will be surrounded by a dynamically defined processor cloud of which the applications are expected to cooperate smoothly. AmI was put forward as one of the major strategic research themes by the IST Advisory Group of the European Commission for the financing structure of the 6th Framework of the European Union [1, 2]. Meanwhile, the first European symposium on AmI has recently been organised and institutions like the MIT and Phillips have published their visions on the matter. Currently, AmI seems to group previously "unrelated" fields such as context dependency, domotics, ubiquitous/pervasive computing, mobility, intelligent buildings and wearable hardware. Early experiments in these fields already seem to indicate that their full development will need a new generation of programming languages that have dedicated provisions to deal with highly dynamic hardware and software constellations. As such, AmI will open up a new "market" for a new generation of programming languages which are designed to write software that is expected to operate in extremely dynamic hardware and software configurations.

The big success of the workshop's first edition at ECOOP 2003 [3] confirmed the feeling that many researchers in the object-oriented community are still interested in object-oriented language design, and moreover, many are interested in languages that move away from Java's main design lines. The goal of this second edition of the workshop was to address object-oriented languages that diverge from Java's doctrine but support a much more dynamic way of constructing software. In the near future, this dynamicity will be required in order to construct software that is highly context-dependent due to the mobility of both the software itself and its users, as is the case in AmI. There is a new future for languages based on Lisp, CLOS, Scheme, Self, Smalltalk and loads of less well-known academic languages. This new generation of programming languages will exhibit a mix of new and old ideas. Many position papers submitted to the workshop support this view.

2 Organisation

This section summarises some organisative aspects of the workshop and gives information about attendance.

The submitted position papers, the invited talk slides, and complementary information about the workshop can be found at the workshop website:
`http://prog.vub.ac.be/~wdmeuter/PostJava04/`

2.1 Call for Contributions

The call for contributions invited researchers to submit a position paper or an essay (6 pages maximum) about new language features or about existing ones that cover solutions to problems that are currently getting relevant in mainstream languages. Provocative and visionary contributions were especially encouraged.

Dynamicity as required by the AmI vision was selected as the common theme of the workshop, i.e. a new context in which we can talk about the object-oriented language features of the future. Hence the "Back to Dynamicity" part in the workshop's title. Topics of interest were formulated as follows:

- agent languages,
- distributed languages,
- actors, active objects,
- delegation,
- mixins,
- prototypes,
- multi-paradigm programming,
- meta-programming and reflection,
- mobile languages,
- (distributed/mobile) virtual machines,
- other exotic dynamic features which could be categorised as OO.

2.2 Organisers

Wolfgang De Meuter
wdmeuter@vub.ac.be

Pascal Costanza
costanza@web.de

Stéphane Ducasse
ducasse@iam.unibe.ch

Richard Gabriel
rpg@dreamsongs.com

Theo D'Hondt
tjdhondt@vub.ac.be

The affiliations of the organisers are mentioned in the title of this report.

The organisers asked the first author to write this report (even though he is not an organiser himself), since he played the role of reporter of the workshop. The cooperation was natural given that the workshop proposal was actually a merge of two earlier proposals, one of which was co-submitted by the first author.

2.3 Format

The workshop started with an invited talk by Dave Thomas (see section 3.1). The presentation of each position paper followed, in 30-minute slots including questions. The more "technical" papers were first in the lineup. The more "philosophical" papers were presented after lunch. A plenary discussion session was held afterwards, and the workshop finished with a short wrap-up / evaluation part.

2.4 Attendance

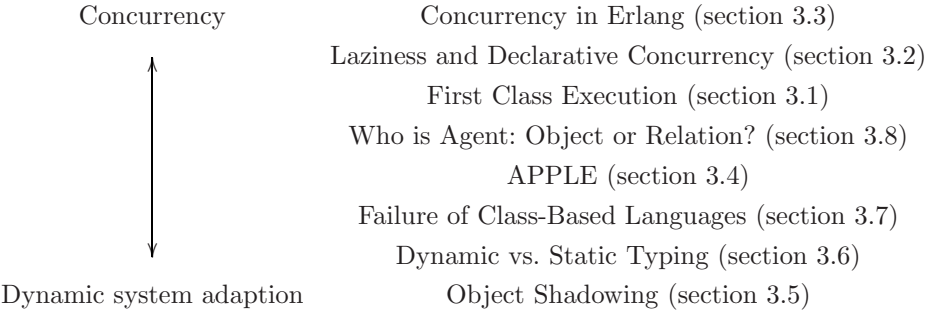
A total of 9 position papers were received. PDF files can be found at the workshop website (see section 4). Of those 9 submissions, 7 authors attended the workshop. Other 13 attendees were present for a total of 20 people:

1. Christopher Anderson, Imperial College London – UK
2. Marie Beurton-Aimar, University of Bordeaux – France

3. Ilia Bider, Ibisoft – Sweden
4. Alex Buckley, Imperial College London – UK
5. Raphaël Collet, UCL – Belgium
6. Marc Conrad, University of Luton – UK
7. Pascal Costanza, University of Bonn – Germany
8. Wolfgang De Meuter, VUB – Belgium
9. Theo D'Hondt, VUB – Belgium
10. Jean-François Gélinas, UQTR – Canada
11. Sebastián González, UCL – Belgium
12. Christian Heinlein, University of Ulm – Germany
13. Jonne Itkonen, University of Jyväskylä – Finland
14. Erik Meijer, Microsoft – USA
15. Sven-Olof Nyström, Uppsala University – Sweden
16. Maximilian Störzer, University of Passau – Germany
17. Eric Tanter, University of Chile – Chile
18. Dave Thomas, Bedarra Research Labs – Canada
19. Richard Torkar, BTH – Sweden
20. Jianguo Zhou, University of Leicester – UK

3 Presentations

This section accounts for the workshop presentations. Two main lines were observed in the contributions this year: concurrency and dynamic system adaption. Some position papers were in one extreme (e.g. concurrency in Erlang) or in the other (e.g. object shadowing), but most of them are influenced by both forces. An approximate classification of the presentations (including Dave Thomas's invited talk) could be:



In the reminder of this section, we hope to give a complementary view of the position papers by describing the presentations rather than the papers themselves. The highlights of the discussions by the participants are marked with a **[Discussion].** label so that they can be easily spotted, although we don't aim at covering each and every nit. The presentation order here is the same as in the workshop.

3.1 Invited Talk: *First Class Execution – Messages and Actors*

Dave Thomas and Brian Barry, Bedarra Research Labs

In his invited talk, Dave Thomas advocated the reification of messages in object-based programming languages. Current commercial OO languages lack support for first class execution (unlike Beta, ABCL etc.). One symptom is that messages have no meta-messages, while classes *do* have meta-classes. A second one is that message dispatch (evaluation) is usually "hidden", i.e. performed using internal machinery hidden in the compiler or virtual machine. Ideally, messages should have first class status. Many applications are considerably simpler if the viewed from the perspective of the message rather than the object/process. This message-centric view, baptised *Message-Oriented Programming* by Dave Thomas [4], is aligned with the ideas presented by Ilia Bider (see section 3.8).

Method dispatch can be abstractly defined in message-oriented programming as: *eval(message, args, sender, receiver)*, where the sending and receiving objects are explicit parameters to the message evaluation in addition to the arguments. A quick comparison shows the differences with two other fundamental approaches:

- Scheme: given the function (method) find the environment.
- Smalltalk: given the environment, find the function (method).
- Message-oriented programming: given the message determine the function (method), knowing the message definition and the value of the sender (environment) and the receiver (environment).

Message-oriented programming is one part of a broader view on how to structure Service Oriented Architectures (SOAs) [5]. The coordination of services (nowadays web services) has proved to be difficult using petri nets or state machines. Dave Thomas turned attention to Carl Hewitt's actor model [6] to tackle this problem. Actors are autonomous and concurrent objects that communicate asynchronously and are intended to be a model of an intelligent person. When an actor receives a message, it executes according to its script and communicates with a well-defined and finite set of other known actors. Actors allow one to model computations as an organisation of communicating active objects and to apply anthropomorphic roles such as Workers, Coordinators, Managers, Couriers, Notifiers, or more application-specific ones like PulseCourier, RadarTrack, TrackManager. Hence, workflow in the system mimics real-world workflow. This *Anthropomorphic Programming* approach allows business processes to be expressed using common organisational design principles.

Anthropomorphic programming was used for process structuring in the Harmony operating system: tasks are assigned personified roles such as the already mentioned Servers, Administrators, Workers, Couriers, Notifiers, etc. Each of these roles has well known pre-defined semantics. Servers must be responsive, so they delegate most of the work. Processes spend most of their life in a "receive any" loop, while Workers do most computations. The Administrator helps organise this. All these tasks are very lightweight (in Harmony, tasks = processes = threads). Harmony is a message-based operating system, with a simple set of primitives: Blocking Send, Blocking Receive, Reply, Create, Terminate and special forms for Non-Blocking Receive and Interrupts.

Harmony was used as a foundation for the Actra project. Actra sought to show how far Smalltalk could be used in the development of complex embedded applications. The Actra project combined Smalltalk, actors and multiprocessors. An actor encapsulates cooperating passive (non-actor) objects. Actors synchronize and communicate by sending messages. Actors execute in pseudo parallel on a single processor and in parallel on multiple processors. Actors have the granularity of lightweight processes (threads/tasks). There are uniform semantics for remote/local processes and these processes have a well defined life cycle. In Actra, programmers create their own application-specific actors by specialising the generic ones. The complete taxonomy of known actors, some generic, many more application specific, creates a vocabulary that populates the programming model and defines its semantics.

[Discussion]. One participant asked why Smalltalk is not considered to have first-class messages, given that the `doesNotUnderstand:` message can be overridden to get an instance of the `Message` class. This was considered a hack by Thomas, rather than a true first-class mechanism. Apart from this, the sender of the message is not available, unless a complex inspection of the runtime stack is performed.

3.2 Laziness and Declarative Concurrency

Raphaël Collet, Université catholique de Louvain (Belgium), raph@info.ucl.ac.be

Raphäel Collet presented an extension of the concurrent declarative framework of the Oz language with *by-need synchronisation*. Oz has a store consisting of constraints over logic variables. There are two operations, *ask* and *tell*. Telling a constraint simply adds it to the store. Asking a constraint makes the thread wait until the store can logically infer it or its negation.

Programming a demand-driven computation in a dataflow concurrent language is easy. The computation is put in a thread and is suspended until another thread manifests its need for the result. The way of manifesting the need is by telling `need(x)` to the store, where `x` is the variable. This constraint does not restrict the possible value of `x`. In this way dataflow synchronisation of multiple threads is achieved.

3.3 Concurrency in Java and Erlang

Sven-Olof Nyström, Uppsala University (Sweden), svenolof@csd.uu.se

Java's threads are rather nicely integrated with the class system. Unfortunately, the implementations use the threads of the underlying operating system, which means that threads are expensive. Many operating systems only allow a very restricted (a few hundred) number of threads. What is even worse is that the behavior of threads depends on the operating system, so that a Java program written for one OS might not work when run on an other OS. The use of OS threads is discouraged by the Erlang designers.

In contrast, all Erlang processes under a specific node share the same OS-process. An Erlang process is a data structure, containing a stack, a heap and a

program counter. The stack and the heap are small at creation, and allowed to grow when necessary, so the minimum size of a process is a few hundred bytes. Erlang can easily support thousands of threads.

[Discussion]. An operating system based on Erlang could be developed, given its reliability and concurrent design.

3.4 APPLE: Advanced Procedural Programming Language Elements

Christian Heinlein, University of Ulm (Germany),
heinlein@informatik.uni-ulm.de

Christian Heinlein started by observing that current programming languages, in particular aspect-oriented languages such as AspectJ, are considerably complex, in contrast with traditional procedural languages, such as Pascal or C, which provided just two basic building blocks: data structures and procedures. The gain in complexity is not reflected proportionally in the gain of expressive power.

He pointed out that there are many features in aspect-oriented languages which could be dropped from the kernel of these languages and implemented in terms of other features. After many conceptual reductions, he argues that "around advice" (i.e. the possibility to freely override an existing procedure, either with completely new code or with code that augments the original code) remains as one of the few essential (i.e. really necessary) mechanisms.

His position was to go back to the starting point of procedural programming languages and extend them in a different direction, not leading to OO or AOP programming, but to "advanced procedural languages" which are significantly simpler than aspect-oriented languages while offering comparable expressiveness and flexibility.

Three points are the key:

1. Replacing statically bound procedures with dynamically overridable procedures (roughly comparable to around advice) covers the whole range of dynamic dispatch strategies usually found in object-oriented languages. Dynamic procedures remain a single, well-defined concept, hardly more complex than traditional procedures.
2. Replacing record types having a fixed set of fields with modularly extensible "open types" and "attributes" (roughly comparable to empty classes extended by inter-type field declarations) covers a wide range of (practically all) OO abstractions. Again, open types constitute a single, well-defined concept which is little more complex than traditional record types.
3. Preserving (resp. (re-)introducing) the module concept of modern procedural languages with clearly defined import/export interfaces and a strict separation of module definitions and implementations, provides support for encapsulation and information hiding.

Instances of open types differ from instances of classes in object-oriented languages such as Java or C++ in two ways. First, their set of attributes is dynamic, again in two ways: because the attributes of a particular type can be

defined in different modules, the set of all attributes is unknown when compiling a single module. Furthermore, since modules containing attribute definitions might be loaded dynamically at run time, the set of all attributes belonging to a type is even unknown at link or program start time. If an attribute is accessed that is not present yet, a well-defined null value is returned for read accesses, while a new attribute/value pair is added for write accesses. Second, the dynamic type of an object, which is equal to its static type immediately after creation, can be changed at run time.

Because procedures are not directly associated with types or objects, such manipulations cannot lead to "message not understood" or other run time type errors, i.e., the system remains statically type-safe.

[Discussion]. If a dynamic procedure is overridden in different modules, the linear "module order" that is uniquely determined by the modules' import relationships, determines the order of the procedure redefinitions.

[Discussion]. Dynamic procedures solve the well-known "expression problem" (it is hard to add both new types and new operations to an existing type hierarchy) in a simple and straightforward manner – so simple that some of the participants found it hard to believe.

3.5 Object Shadowing – A Key Concept for a Modern Programming Language

Marc Conrad, Tim French, and Carsten Maple, University of Luton (UK), {marc.conrad, tim.french, carsten.maple}@luton.ac.uk

Shadow objects mask one or more methods in a target object (the "shadowed" object). A shadow is applied at run-time rather than compile-time, in response to dynamic needs. Every message sent to the shadowed object is processed by the shadow, if the shadow defines it, or otherwise it is passed to the shadowed object as if there were no shadow in between. More characteristics of the shadow mechanism are described in the position paper. The shadow mechanism has for long been available in LPC (a highly pragmatic language used for text based computer games, see http://wwwlysator.liu.se/mud/lpc.html) but has not been evaluated academically so far.

Marc Conrad presented possible application areas for this mechanism:

1. Deprecated Methods: a shadow system could help the developer of a library to separate an object into two parts. The actual, official version of the object where the deprecated methods have been removed and a collection of shadows that implement deprecated methods. The shadows avoid breaking existing (legacy) clients.
2. Prototyping: a shadow could be used to adapt the behaviour of objects in a library, in situations where the objects cannot be directly manipulated because the library has been bought from an external supplier or because of copyright issues.
3. Reclassification: reclassification and a special case of it, dynamic inheritance, is the process of changing the class of an object at run-time. The main goal

in reclassification is to modify the behaviour of an object. This could be achieved by the application of a shadow to the object. For example a player having a temporal "frog shadow" might change its nature by replacing it with a "prince shadow" (but it remains a player).

4. Interclassing: the basic principle in interclassing is the insertion of a new class in an existing inheritance hierarchy. Suppose an existing hierarchy has `Parallelogram` as a parent of `Square`, and that `Rectangle` is introduced as a specialization of `Parallelogram`. Now, `Square` should inherit from `Rectangle`, and to this end, one could shadow `Square` with a `SquareShadow`, which inherits from `Rectangle`. All the clients of `Square` will now see the behaviour of `Rectangle`, even if `Square` itself is not modified.
5. Inheritance and Specialization: A shadow could even be used to emulate inheritance. In particular, a programming language that has shadows as a first class feature and derives inheritance as a special application of shadows can be envisioned. However the presenter expressed fears that this particular idea may be too wild and not relevant for any practical implementation.

At the end of his talk Marc Conrad raised the (provocative) question, why such a useful feature is not available in mainstream languages?

[Discussion]. It was pointed out that the last feature, emulation of inheritance by shadows (the feature that has been considered as too esoteric by the presenter), could be the most useful, as it would allow the implementation of different *roles* that an object may have during its lifetime.

[Discussion]. The class `java.lang.reflect.Proxy` of Java may serve a similar purpose as a shadow but it was pointed out that this Java class works only for interfaces and cannot shadow classes or objects.

There was some agreement in the audience that shadows may be a useful feature and the question why it is not available in mainstream language is valid.

3.6 Dynamic Versus Static Typing – A Pattern-Based Analysis

Pascal Costanza, Universität Bonn (Germany), costanza@web.de

The main point Pascal Costanza made is that statically typed languages sometimes force solutions which in dynamic typed languages could be expressed more naturally or are even available by default. Furthermore, statically-typed languages introduce new sources of potential bugs, contrary to conventionally perceived wisdom. He presented three examples of these situations, in the form of patterns. Java was chosen as a representative example of a statically typed language. The three examples shown were the following:

Statically Checked Implementation of Interfaces. When implementing an interface in Java, usually the programmer "fills in" the required methods with dummy bodies (e.g. which simply return `null`, `0` or `false`), so that the program compiles. Clean compilation is needed for incremental development of the class. The problem with these dummy implementations is that they are a potential bug which might be hard to find later on. The solution is to throw dynamically

checked exceptions indicating that the involved methods are currently not implemented, instead of providing a dummy implementation, as in the following example:

```
public class FileCharSequence implements CharSequence {
   public FileCharSequence() {...}
   public char charAt(int index) {...}
   public int length() {...}
   public CharSequence subSequence(int start, int end) {
      throw new UnsupportedOperationException
         ("FileCharSequence.subSequence not implemented yet.");
   }
}
```

But the main point is, dynamically typed languages do exactly this by default: the sending of a message which is not implemented will raise an exception, à la Smalltalk's "message not understood".

[Discussion]. The problem could be seen as an IDE support issue rather than a programming language flaw: for example in Eclipse, one can configure the IDE so that code similar to the above is generated automatically. But by default, Eclipse generates dummy code.

Statically Checked Exceptions. This issue is better explained with an example. Assume that a class performs some heavy computations based on statistical data. It should be possible to deploy this class by itself, in which case the data is read from a local file, or else it will fetch the data from a remote file via an RMI service. In the latter case, this class needs to deal with the statically checked RMI exceptions. The problem is that declaring RMI exceptions is not appropriate to the abstraction provided by the statistical class.

The solution is to use dynamically checked exceptions. They are passed on by any code without the need to even mention the exceptions. A class only needs to explicitly deal with exceptions it is concerned with. The point in favour of dynamically typed languages with exception handling mechanisms, is that they have this functionality by default.

[Discussion]. In the case of Java, the RMI exception can be wrapped with a `RuntimeException` so that the exception declaration can be omitted.

Checking Feature Availability. Checking if a resource provides a specific feature and actually using that feature should be an atomic step in the face of multiple access paths to that resource. Otherwise, that feature might get lost in between the check and the actual use. Example:

```
System.out.println("Name: " + person.getName());
if (person instanceof Employee) {
   System.out.println("Employer: " +
      ((Employee)person).getEmployer().getName();
}
```

If the value of `persons` is changed by another thread between the type check and the type cast, an exception will be raised. Static typing promotes the notion that the availability of a particular feature should be checked before it is actually used. For example, fields and methods can be regarded as features of classes.

The solution is to make the check and the use an atomic step:

```
System.out.println("Name: " + dilbert.getName());
try {
   System.out.println("Employer: " +
      ((Employee)dilbert).getEmployer().getName();
} catch (ClassCastException e) {
   // do nothing
}
```

The point in this example is, dynamically typed languages throw "message not understood" errors by default. One only has to catch them instead of `ClassCastException`. Apart from that difference, the resulting behavior is the same.

[Discussion]. Some found that this problem is related to concurrent execution and is not inherent to static typing.

3.7 The Unavoidable Failure of Class-Based Languages in the Processor Cloud Era

Sebastián González, Wolfgang De Meuter, Kim Mens, Theo D'Hondt, Université catholique de Louvain (Belgium) and Vrije Universiteit Brussel (Belgium), {sgm, kim.mens}@info.ucl.ac.be, {wdmeuter, tjdhondt}@vub.ac.be

The point made by this presentation was that class-based languages are not adequate for the programming of the so-called "processor clouds", i.e. ad-hoc mobile networks of wirelessly interconnected computers where nodes can enter and leave the network at any moment, for instance if a user enters or leaves a certain building. The use of object-based languages without the concept of *class* are advocated as a good alternative. The main problem with classes is that they are a *universal*, *stateful* resource sharing mechanism, which is a bad combination of ingredients for open distribution (processor clouds). For example, a node containing the same class as another node may appear in the network, but with a different value for a class variable or a different method implementation; this is an unsolvable conflict since none of the two versions is the "right" one. Other problems with classes mentioned were the fact that classes grow monotonically, getting deprecated methods, which waste resources (storage space, network bandwidth) in a context where resources are scarce: mobile computing. Another problem is that idiosyncratic behaviour is even more important in mobile computing, which is harmed by the usage of classes. A last problem was mentioned: upon sending a message to a node, a class and its superclass(es) need to be sent along, and if the language is statically typed, argument-type, result-type and exception-type classes need to be sent also.

3.8 Who Is Agent: Object or Relation?

By Ilia Bider, IbisSoft (Sweden), ilia@ibissoft.se

Starting from the observation that any object-oriented system can be considered as consisting of objects and relations between them, Ilia Bider proposed a rather different perspective for the role these objects and relations play. In his programming model, the centre of attention is on the relations. This contrasts with traditional OO programming where attention is mainly put on objects, and relations are usually simple pointers. In the new perspective, while objects are passive, relations are active. The relations or *connectors* represent laws which are to be maintained throughout the system. Whenever the state of an object changes (e.g. because of an external user action), the relations connected to that object are verified. If necessary, these relations change the state of other connected objects in order to bring the system back to an acceptable state. Changes are propagated in this way through object networks. The relations are perceived as the "agents" in the system: the active elements which perform computation and evolve its state. This motivates the title of his position paper.

The described object-connector model is proposed as a method of distributed programming. The whole system is expressed in terms of local laws, with communication and execution control being automatically provided by the environment.

[Discussion]. The workshop participants found the programming model quite different from what they are used to in standard OO programming. No flaws or problems were pointed out.

4 Related Work

The submitted position papers, Dave Thomas's slides, and complementary information about the workshop can be found at the workshop website:
`http://prog.vub.ac.be/~wdmeuter/PostJava04/`

The website for the previous edition of the workshop is located at:
`http://prog.vub.ac.be/~wdmeuter/PostJava/`

And there is already a followup event planned at OOPSLA 2004, the 1st Workshop on the Revival of Dynamic Languages:
`http://pico.vub.ac.be/~wdmeuter/RDL04/index.html`

This workshop was inspired by Richard Gabriel's Feyerabend events at past OOPSLAs and ECOOPs. The home page of the Feyerabend Project is located at:
`http://www.dreamsongs.com/Feyerabend/Feyerabend.html`

References

1. Ducatel, K., Bogdanowicz, M., Scapolo, F., Leijten, J., Burgelman, J.C.: Scenarios for ambient intelligence in 2010. Technical report, EC Information Sociecty Technologies Advisory Group (ISTAG) (2001) Available [2004-07-12] at http://www.cordis.lu/ist/istag-reports.htm.

2. Shadbolt, N.: Ambient intelligence. IEEE Intelligent Systems **18** (2003) 2–3
3. De Meuter, W., Ducasse, S., D'Hondt, T., Madsen, O.L.: Object-oriented language engineering for the post-java era. In Buschmann, F., Buchmann, A.P., Cilia, M., eds.: Object-Oriented Technology: ECOOP 2003 Workshop Reader. Volume 3013., Springer-Verlag (2004)
4. Thomas, D.: Message oriented programming. Journal of Object Technology **3** (2004) 7–12 Available [2004-07-12] at http://www.jot.fm/issues/issue_2004_05/column1.
5. Thomas, D., Barry, B.: Using active objects for structuring service oriented architectures. Journal of Object Technology **3** (2004) 7–14 Available [2004-07-12] at http://www.jot.fm/issues/issue_2004_07/column1.
6. Hewitt, C.E.: Viewing control structures as patterns of passing messages. Journal of Artificial Intelligence **8** (1977) 323–364