

# Announcements: an implementation of implicit invocation

JOHN ALAN McDONALD

MARK NIEHAUS \*

Dept. of Computer Science and Engineering

and

Dept. of Statistics,

University of Washington

October, 1991

## Abstract

This report describes the Announcements module, which provides a mechanism for implicit invocation (i.e. broadcast message sending) for Common Lisp programs.

## 1 Introduction

The Announcements module provides a mechanism for maintaining simple dependencies between objects through message broadcasting, similar to that used in the Field [8] and Forest [1] environments. It is derived from a Common Lisp implementation of Events/Mediators [12] by Mark Niehaus for the Prism system [3]. It is in part motivated by dissatisfaction with some aspects of earlier work by one of the authors in the Antelope system [4] and the Plot Windows system of Stuetzle [11, 10].

The Announcements module is a component of a system called Arizona, now under development at the U. of Washington. Arizona is intended to be a portable, public-domain collection of tools supporting scientific computing, quantitative graphics, and data analysis, implemented in Common Lisp and CLOS (the Common Lisp Object System) [9]. This document assumes the reader is familiar with Common Lisp and CLOS. An overview of Arizona is given in [5] and an introduction to the current release is in [7].

Like events in Field and Forest, Announcements can be thought of as a mechanism for *implicit invocation* as discussed in [2, 13]. The basic idea is to allow an object (the *announcer*) to notify other objects (the *audience*) of a change of state (or some other event) without the announcer having to know which objects need to be notified of which events.

The interface to Announcements supplies only the Events portion of an Events/Mediators mechanism, which means that it can be used to implement simple one-way dependencies, but not the more complicated relationships that would require some version of Mediators. The reason for this is that we feel we understand the Events part and can produce a stable, self-contained implementation that should be useful on its own in a wide variety of applications. In contrast, our understanding of what would be a good set of broadly useful Mediators, or what an appropriate protocol for an abstract Mediator type might be, is much less secure. We expect that Announcements will be most useful when used together with application-specific Mediators.

---

\*This work was supported in part the Dept. of Energy under contract FG0685-ER25006 and by NIH grant LM04174 from the National Library of Medicine.

## 2 How to use Announcements

Conceptually, an Announcement object represents a type of event that might occur in a number of announcer objects. The Announcement object keeps track of an audience for each potential announcer and sees that each member of the audience is properly notified when the announcement takes place.

To make this a little more concrete, consider the case of a direct-manipulation graph browser. The browser described in [6] defines a simple protocol so that it can be used with a broad class of data structures representing graphs. Any use of this browser involves two modules that implemented more-or-less independently: the interactive browser and the data structure representing the underlying graph. The dependency that we want to represent and maintain with Announcements is that the browser display should automatically update when nodes or edges are added to or deleted from the underlying graph.

### 2.1 Creating Announcements

In normal use, a globally accessible announcement `clay:clay-object` is defined with a call to `an:defannouncement`, which is a defining form similar to `defclass`. In particular, it is like `defclass` in the sense that re-evaluating `an:defannouncement` for a given name does not create a new Announcement object, but simply updates the lambda list and documentation if they have changed.

It may sometimes be useful to create Announcements in a function call, using `an:make-announcement`. Announcements created with `an:make-announcement` are not automatically accessible by name. An Announcement can be made globally accessible by name with a `setf` of `an:find-announcement`, which will replace the existing announcement with the given name, if there is one.

A simple way to handle the graph browser example is to provide an Announcement that corresponds to any change in the state of the underlying graph, as follows:

```
(an:defannouncement :subject-changed ())
```

This `:subject-changed` announcement would most likely be defined as part of a general toolkit for direct manipulation interfaces, and not as part of the graph browser itself.

One could get higher performance, at the cost of a loss in simplicity, by defining a more specialized Announcement interface, eg., defining Announcements for `:nodes-added`, `:nodes-deleted`, etc. Alternatively, one could define `:subject-changed` with a lambda list that would allow arguments to be passed that specify exactly how the underlying graph had changed.

In the example above, the announcement's name is a keyword. An announcement name can, in fact, be any symbol. However, keywords may be a good idea when the announcement is not clearly part of any particular package/module and when potential announcers and audience members may reside in many different packages.

### 2.2 Joining an Audience

The announcer of an announcement can be any lisp object, though the normal, intended use is for it to be an instance of a CLOS class. Any object can join the audience for a particular announcer-announcement pair with a call to `an:join-audience`. What happens when a particular member of the audience is notified is determined by the notification function supplied as the last argument to `an:join-audience`. If `an:join-audience` is called more than once with the same announcer, announcement, and audience member, the notification function will be that provided with the last call.

For the graph browser example, the browser object will need to join the audience for announcements of changes to the state of its subject graph:

```
(an:join-audience (subject graph-browser) :subject-changed
                  graph-browser #'subject-state-changed)
```

The `subject` of the browser is the graph data structure that it displays. `Subject-state-changed` is a generic function that updates the display.

Such calls to `an:join-audience` would typically occur in the initialization code for the browser object, or in code that changes which graph a browser is displaying.

In other systems for implicit invocation, like `Field` [8] and `Forest` [1], some sort of pattern matching is done to determine who gets notified of which announcements. In the `Announcements` module, the pattern matching is limited to testing for equality of announcer, announcement, and audience member. The sense of equality that's used is `eq`.

If a member of the audience no longer wishes to be notified of announcements by the announcer, it can remove itself from the audience with a call to `an:leave-audience`. For example, if a browser were to change which graph it displayed, it would first have to do something like:

```
(an:leave-audience (subject graph-browser) :subject-changed graph-browser)
```

## 2.3 Announcing

Any announcer (any lisp object) can `an:announce` any defined announcement, which results in every member of the audience for that announcer-announcement pair being notified. Nothing happens if no object is in the audience for the given announcement by the given announcer. Notification means that the function argument supplied to `an:join-audience` is applied to the corresponding member of the audience, together with whatever other arguments are supplied to `an:announce`. The arguments that are supplied in the call to `an:announce` must be consistent with the lambda list supplied in `an:define-announcement` or `an:make-announcement`.

In our simple example, the `:subject-changed` announcement takes no arguments, so a graph object would announce thus:

```
(an:announce graph :subject-changed)
```

### 3 Reference Manual

an:announce	Generic Function
-------------	------------------

**Documentation:**

Notify all members of the audience of the given <announcer> and <announcement>.

**Usage:**

(an:announce announcer announcement &rest args)

**Arguments:**

announcer — T  
 announcement — (Or Symbol An:Announcement)  
 args — T

**Returns:**

t

**Source:** /belgica-2g/jam/az/announcements/announcements.lisp

An:Announcement	Class
-----------------	-------

**Documentation:**

An announcement maintains an audience (for each announcer) to notify when it is announced by an announcer.

**Usage:**

(typep x 'An:Announcement)

**Parents:**

Standard-Object

**Source:** /belgica-2g/jam/az/announcements/announcements.lisp

:Announcements	Package
----------------	---------

**Usage:**

(in-package :Announcements)

**Source:** /belgica-2g/jam/az/announcements/package.lisp

an:audience	Generic Function
-------------	------------------

**Documentation:**

<Audience> returns a list of items for any announcer — announcement pair. There is one item in the list for each member of the audience. Each item is a list of length 2; the first item in the list is the member of the audience and the second item is a function (or function name). When an

announcement is made by a given announcer, each member of the audience is notified by applying the function to the member and whatever additional arguments were passed to <announce>.

**Usage:**

(an:audience announcer announcement)

**Arguments:**

announcer — T  
announcement — (Or Symbol An:Announcement)

**Returns:**

audience-list — List

**Source:** /belgica-2g/jam/az/announcements/announcements.lisp

(setf an:audience)	Generic Function
--------------------	------------------

**Documentation:**

<setf audience> associates a new audience list with a given announcer — announcement pair. One does not often want to completely change an audience list, but the setf method needs to be available for use by other functions that make smaller modification to an audience list.

**Usage:**

(setf (an:audience announcer announcement) new-audience)

**Arguments:**

new-audience — List  
announcer — T  
announcement — (Or Symbol An:Announcement)

**Returns:**

new-audience

**Source:** /belgica-2g/jam/az/announcements/announcements.lisp

an:defannouncement	Macro
--------------------	-------

**Documentation:**

If an announcement with the given name does not exist, create one and make it globally accessible by name. Otherwise update the <lambda-list> and <documentation> of the existing announcement object with the given name. It is an error to change the <lambda-list> so that it is incongruent with existing notification function, but no error will be signaled at <defannouncement> time.

**Usage:**

```
(an:defannouncement name lambda-list (&key (documentation )))
```

**Returns:**

An:Announcement

**Source:** /belgica-2g/jam/az/announcements/announcements.lisp

an:find-announcement	Function
----------------------	----------

**Documentation:**

Find the announcement with a given <name>, returning nil if one does not exist.

**Usage:**

```
(an:find-announcement name)
```

**Arguments:**

name — Symbol

**Returns:**

(Or An:Announcement Null)

**Source:** /belgica-2g/jam/az/announcements/announcements.lisp

(setf an:find-announcement)	Setf
-----------------------------	------

**Documentation:**

Associate <announcement> with <name>. <Name> must be <eq> to the <announcement>'s name. This setf function must be used with care.

**Usage:**

```
(setf (an:find-announcement name) announcement)
```

**Source:** /belgica-2g/jam/az/announcements/announcements.lisp

an:join-audience	Generic Function
------------------	------------------

**Documentation:**

If <audience-member> is not already represented in the audience for the <announcer> <announcement> pair, then add a new item. Otherwise, just update the notification function.

**Usage:**

```
(an:join-audience announcer announcement audience-member notification-function)
```

**Arguments:**

announcer — T  
announcement — (Or Symbol An:Announcement)  
audience-member — T  
notification-function — (Or Symbol Function)

**Returns:**

t

**Source:** /belgica-2g/jam/az/announcements/announcements.lisp

an:leave-all-audiences	Function
------------------------	----------

**Usage:**

(an:leave-all-audiences audience-member)

**Arguments:**

audience-member — T

**Returns:**

t

**Source:** /belgica-2g/jam/az/announcements/announcements.lisp

an:leave-audience	Generic Function
-------------------	------------------

**Documentation:**

Remove the item corresponding to <audience-member> from the audience list for the <announcer> <announcement> pair.

**Usage:**

(an:leave-audience announcer announcement audience-member)

**Arguments:**

announcer — T  
announcement — (Or Symbol An:Announcement)  
audience-member — T

**Returns:**

t

**Source:** /belgica-2g/jam/az/announcements/announcements.lisp

an:leave-audience-for-all-announcers	Generic Function
--------------------------------------	------------------

**Documentation:**

Remove all the items corresponding to <audience-member> from the audience list for any announcer and this <announcement>.

**Usage:**

(an:leave-audience-for-all-announcers announcement audience-member)

**Arguments:**

announcement — (Or Symbol An:Announcement)  
audience-member — T

**Returns:**

t

**Source:** /belgica-2g/jam/az/announcements/announcements.lisp

an:make-announcement	Function
----------------------	----------

**Documentation:**

Make an announcement object with the specified attributes. This announcement object will not be automatically installed in any global tables. To make it globally accessible, and overwrite existing announcement with name <name>, use

(setf (find-announcement name) announcement-object).

**Usage:**

(an:make-announcement &key name lambda-list documentation)

**Arguments:**

name — Symbol  
lambda-list — List  
documentation — String

**Returns:**

An:Announcement

**Source:** /belgica-2g/jam/az/announcements/announcements.lisp

## References

- [1] David Garlan and Ehsan Ilias. Low-cost, adaptable integration policies for integrated environments. In *Proceedings of ACM SIGSOFT'90: Fourth Symposium on Software Development Environments*, pages 1–10, December 1990.
- [2] David Garlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *Proceedings of VDM'91: Formal Software Development Methods*, 1991.
- [3] Ira Kalet, Christine Sweeney, and Jonathan Jacky. Software design for interactive graphic radiation treatment simulation systems. In *Proceedings of the Fourteenth Annual Symposium on Computer Applications in Medical Care*, pages 594–598, Washington, D.C., November 1990. IEEE Computer Society Press.
- [4] John Alan McDonald. Antelope: data analysis with object-oriented programming and constraints. In *Proc. of the 1986 Joint Statistical Meetings, Stat. Comp. Sect.*, 1986.
- [5] John Alan McDonald. An outline of Arizona. In *Computer Science and Statistics: Proc. 20th Symp. on the Interface*, pages 282–291, Washington, D.C., 1988. ASA.
- [6] John Alan McDonald. A simple graph browser. Technical report, Dept. of Statistics, U. of Washington, October 1991.
- [7] John Alan McDonald and Michael Sannella. Arizona overview and notes for release 1.0. Technical report, Dept. of Statistics, U. of Washington, October 1991.
- [8] Steven P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66, July 1990.
- [9] G.L. Steele. *Common Lisp, The Language*. Digital Press, second edition, 1990.
- [10] Werner Stuetzle. Design and implementation of plot windows. In *Proceedings of the Statistical Computing Section of the American Statistical Association*, pages 32–40, 1987.
- [11] Werner Stuetzle. Plot windows. *JASA*, 82:466–475, 1987.
- [12] Kevin J. Sullivan and David Notkin. Reconciling environment integration and component independence. In *SIGSOFT'90: Fourth Symposium on Software Development Environments, Irvine CA*, pages 208–225, June 1990.
- [13] Kevin J. Sullivan and David Notkin. Behavioral relationships in object-oriented analysis. Technical Report 91-09-03, Dept. of Computer Science and Engineering, U. of Washington, 1991.