Open Microscopy Environment

Publisher-Subscriber
Software Design Document

June 2003

Andrea Falconi

Swedlow Lab — MSI/WTB Complex
University of Dundee

a.falconi@dundee.ac.uk

DISCLAIMER OF WARRANTY

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the

```
Free Software Foundation, Inc.,
59 Temple Place, Suite 330,
Boston, MA  02111-1307  USA
```

Contents

1. Introduction 4
   1.1. Requirements 4
   1.2. Solution outline 4
   1.3. Document overview 5

2. Object Model 6
   2.1. Structure 6
   2.2. Dynamics 8

3. Usage 11

References 15

1. Introduction.

This document details the design of our implementation of the Publisher-Subscriber design pattern [POSA1]. Relative test cases are described in a separate document.

Before diving into detailed design, let's briefly outline the required functionality and features. After that, we'll also give an outline of the solution, which is fully described in the next sections.

1.1. Requirements.

We need a general event-propagation mechanism so that:

- Objects can monitor the occurrence of selected events at a source object.
- The source object is not coupled to the identity and number of the monitoring objects, which are only known at run-time.
- Explicit polling of the source object by monitoring objects is avoided.

1.2. Solution outline.

The solution of the above requirements is easily found in the Publisher-Subscriber design pattern [POSA1], also known as Observer [GoF95]. In our implementation a base abstract class, *Publisher*, provides the means for objects implementing the *Subscriber* interface — the monitoring objects — to register interest in some given events. Those events are specified by a concrete *Publisher* — which "publishes" the events. These published events can be fired by the concrete *Publisher* itself or by another event source that is connected to the concrete *Publisher*. When such an event is fired, the *Publisher* notifies every *Subscriber* object that registered interest in that event type.

Subscriber objects are only notified of events that they selected among the published ones and don't have to poll the concrete *Publisher* to find out about such occurrences. Moreover, the concrete *Publisher* doesn't even get to know about *Subscriber* objects and the abstract *Publisher* interacts with the monitoring objects only through the *Subscriber* interface. The actual number of monitoring objects — which may dynamically change at run-time as *Subscriber* objects register interest — is immaterial to the *Publisher.*

1.3. Document overview.

The following sections in this document will deal with:

- *Object Model*: The core of this document, depicting both the static and dynamic model of the software in terms of objects.
- *Usage*: How to extend the Publisher-Subscriber built-in abstract classes in order to get concrete *Publisher* and *Subscriber* objects to work together.

Readers that are not interested in design internals, should just read the *Usage* section, which features a Perl API-doc perspective.

UML [OMG01] diagrams are extensively used throughout this document — with the exception of the *Usage* section — to precisely depict design. Even though all presented diagrams are commented out and many of them are quite self-explanatory, in order to understand in full the semantics of the diagrams a certain familiarity with UML is necessary. Those that are unfamiliar with UML may want to keep a reference at hand, such as [BRJ00].

2. Object Model.

This section describes both the static and dynamic model of the software in terms of objects. Classes and relationships are discussed in the static model. The dynamic model addresses the collective behavior of those elements.

2.1. Structure.

We describe here the static structure, encompassing classes and relationships. Responsibilities, roles and collaborators of each element are detailed.

The Publisher-Subscriber abstract classes — *Publisher*, *Event* and *Subscriber* — provide the infrastructure to allow objects to monitor the occurrence of selected events at a source object. A concrete *Publisher* specifies what events can be monitored by concrete *Subscriber* objects.

The following UML class diagram presents the structure, which is then further detailed, specifying the responsibilities, roles and collaborators of each element.

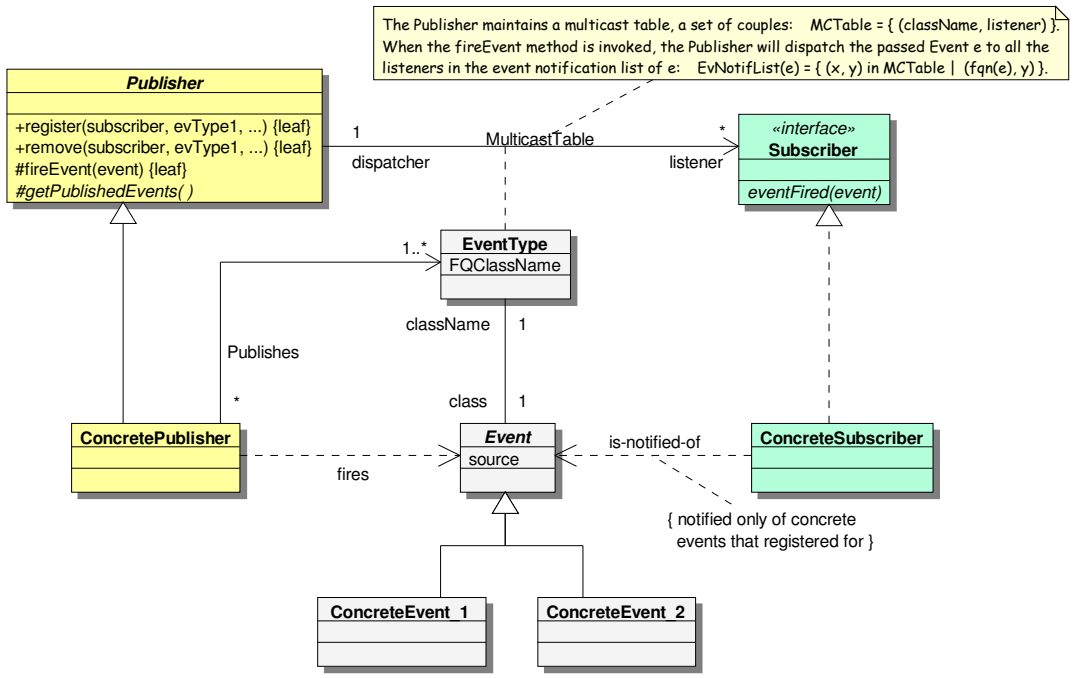

Fig 2-1: Static structure. The leaf constraint denotes non-polymorphic operations (no override).

The classes in the above diagram are detailed below.

Event

Abstract base class to generally represent an event. Its *source* field stores a reference to the object that fired the concrete *Event*. That is usually a concrete *Publisher*, but could also be another event source connected to the concrete *Publisher*.

Subscriber

This interface defines how monitoring objects can be notified of events at a source object. A concrete *Subscriber* has to implement the *eventFired* callback method, which is invoked to dispatch every occurrence of the events that the concrete *Subscriber* registered for. This method has only one argument, a concrete *Event*.

Publisher

Abstract base class that frees concrete *Publisher* objects from having to maintain the infrastructure for event notification. In fact, the *Publisher* already maintains a multicast table, a set of couples:

```
MCTable = { (className, listener) }
```

where *className* is the fully qualified name (Perl package) of a subclass of *Event* and *listener* is a concrete *Subscriber*. When the *fireEvent* method is invoked, the *Publisher* will dispatch the passed *Event e* to all the listeners in the event notification list of *e*:

```
EvNotifList(e) = { (x, y) in MCTable |  (fqn(e), y) }
```

where *fqn(e)* is the fully qualified class name of *e*.

A concrete *Publisher* has to implement the *getPublishedEvents* method to return a list of fully qualified class names of the events that publishes. The *Publisher* uses this list to build the multicast table. Concrete *Subscriber* objects can only register for events that are in this list — through the *register* method. If a concrete *Subscriber* wants to be removed from some event notification lists, then it has to call the *remove* method.

The following UML object diagram is a state snapshot after a concrete *Publisher* — *ConcretePublisher*, which publishes *ConcreteEvent_1* and *ConcreteEvent_2* — has been created and two concrete *Subscriber* objects have registered for some of the published events. Specifically, *s1* — an instance of *ConcreteSubscriber* — registered for both of the above events, as *s2* — an instance of *AnotherConcreteSubscriber* — only registered for *ConcreteEvent_2*.

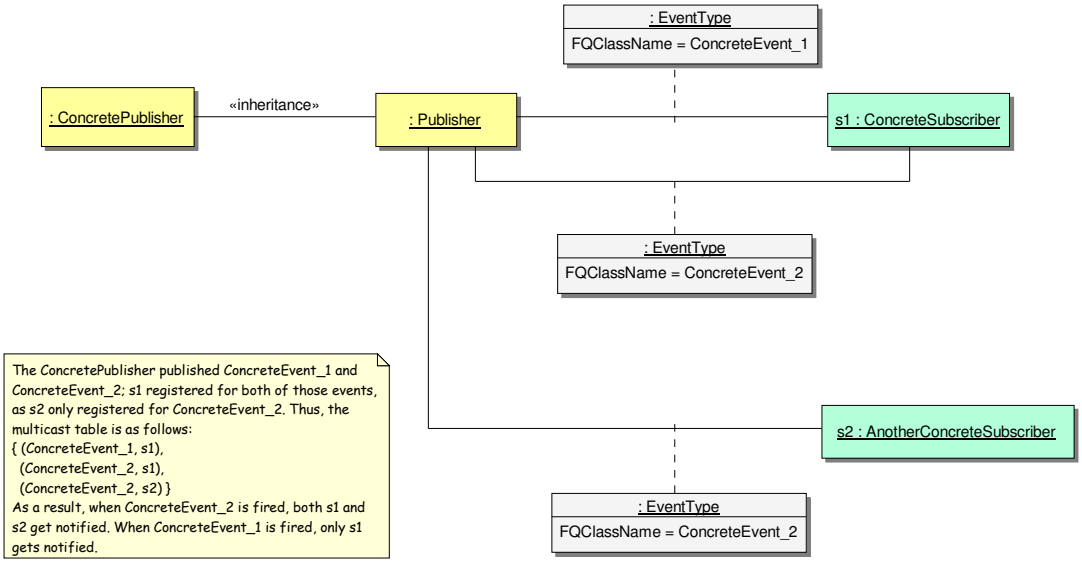

Fig 2-2: State snapshot. The inheritance stereotype means class inheritance.

2.2. Dynamics.

We now focus on the collective behavior of those elements described in the static model. The Publisher-Subscriber classes have to collaborate in order to:

- Publish concrete events.
- Register/remove to/from event notification lists.
- Dispatch events.
- Monitor events.

Every concrete *Publisher* collaborates with its abstract parent to specify the published events. The concrete *Publisher* implements the *getPublishedEvents* method to return a list of fully qualified class names of the events that publishes. The *Publisher* uses this list to build the multicast table at creation time.

Concrete *Subscriber* objects can only register for events that are in this list — through the *register* method of *Publisher*. Moreover, a *Subscriber* may not register twice for the same event — this would cause the *Subscriber* to be notified twice for every occurrence of that event, so no matter how many times a *Subscriber* tries to register for an event, it will only be added once to the corresponding event notification list.
If a concrete *Subscriber* wants to be removed from some event notification lists, then it has to call the *remove* method of *Publisher*.

Event dispatching and monitoring are illustrated by the following interaction scenario:

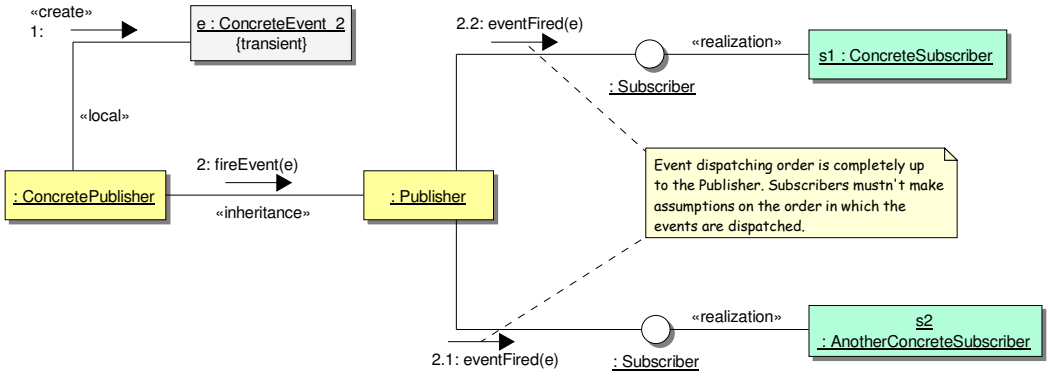

Fig 2-3: Dispatching and monitoring. Notice the use of stereotyped links. The realization stereotype is used to mean the fact that concrete subscribers expose the methods defined by the Subscriber interface because of the realization relationship. The inheritance stereotype means that Publisher is visible through class inheritance. The local stereotype marks links that are method-scoped. Also notice the use of the transient tagged value to denote objects that are in existence only for the duration of the interaction.

We assume that a concrete *Publisher* — *ConcretePublisher*, which publishes *ConcreteEvent_1* and *ConcreteEvent_2* — has been created and two concrete *Subscriber* objects have registered for some of the published events. Specifically, *s1* — an instance of *ConcreteSubscriber* — registered for both of the above events, as *s2* — an instance of *AnotherConcreteSubscriber* — only registered for *ConcreteEvent_2*.

Thus, when the *ConcretePublisher* object fires *ConcreteEvent_2*, the *Publisher* dispatches that event instance to all the listeners in the corresponding event notification list, which, in this case, happens to contain *s1* and *s2*. In contrast, if *ConcreteEvent_1* was fired, then only *s1* would be notified.

Notice that the actual identity of the concrete *Subscriber* objects is immaterial to the *Publisher*, as those objects are only known through the *Subscriber* interface. Their number is irrelevant as well.

A final note. Some trivial issues (such as routine initialization code, constructors and destructors, accessors and mutators or, in general, features that are required to write working source code, but that can be trivially derived from this object model) haven‘t been addressed by this object model explicitly. Those are routine programming tasks that would add little to the semantics of this model — but would inflate this document quite a bit, and for this reason are omitted.

3. Usage.

This section explains how to extend the Publisher-Subscriber built-in abstract classes — *Event*, *Publisher* and *Subscriber* — in order to get concrete *Publisher* and *Subscriber* objects to work together. We explain this through a general-purpose example in Perl API-doc style.

So, say you have a given class that fires some interesting events that need to be monitored. Then, all you have to do is:

- Represent events by sub-classing *Event*.
- Sub-class *Publisher* in order to "publish" those events.
- Have the monitoring classes implement the *Subscriber* interface.

At run-time, concrete *Subscriber* objects can register interest in some of the published events — calling the *register* built-in method of *Publisher*. Whenever the concrete subclass of *Publisher* fires one of the published events — calling the *fireEvent* built-in method of *Publisher* — the abstract parent dispatches that event to all currently registered *Subscriber* objects.

Now, let's assume that the concrete *Publisher* fires two events, which we decided to represent with the classes *ConcreteEvent_1* and *ConcreteEvent_2*. Here's how to code *ConcreteEvent_1*:

```
package MyPkg::ConcreteEvent_1;

use OME::Util::PubSub::Event;
use base qw(OME::Util::PubSub::Event);

sub new {
    my $class = shift;
    my $self = OME::Util::PubSub::Event->new();       # inherit data

    # … add whatever needed …

    bless($self, $class);           # bless as ConcreteEvent_1
    return $self;
}

# … add whatever needed …
```

You could code up *ConcreteEvent_2* pretty much the same as the above. Notice that super-class data has to be explicitly inherited because Perl doesn't do it for you. Concrete

Event classes inherit the *setSource* and *getSource* methods. You use the first one to set a reference to the object that fired the event and the second one to retrieve that reference.

A concrete *Publisher* has to sub-class *Publisher* and implement the *getPublishedEvents* protected class method to return a reference to a list of fully qualified class names (Perl packages) of the events that publishes. Moreover, whenever the concrete *Publisher* detects any published event, it has to create an instance of that event, set the event source and invoke the *fireEvent* protected method (inherited from the abstract parent) passing the event instance. So, in our case, we have:

```
package MyPkg::ConcretePublisher;

use MyPkg::ConcreteEvent_1;
use MyPkg::ConcreteEvent_2;
use OME::Util::PubSub::Publisher;
use base qw(OME::Util::PubSub::Publisher);

sub new {
    my $class = shift;
    my $self = OME::Util::PubSub::Publisher->new();  # inherit data

    # … add whatever needed …

    bless($self, $class);          # bless as ConcretePublisher
    return $self;
}

sub getPublishedEvents {
    my $class = shift;
    my @ev = ('MyPkg::ConcreteEvent_1', 'MyPkg::ConcreteEvent_2');
    return \@ev;
}

sub someMethod_1 {
    my $self = shift;

    # we detected an occurrence of ConcreteEvent_1

    my $e = MyPkg::ConcreteEvent_1->new();
    $e->setSource($self);  # source may well be some other object though
    $self->fireEvent($e);

      . . .
}

sub someMethod_2 {
    my $self = shift;
    my $e = MyPkg::ConcreteEvent_2->new();
    $e->setSource($self);
    $self->fireEvent($e);
    return;
}
```

Again, notice the explicit data inheritance. Also notice that the event source may well be some other object connected to the concrete *Publisher* instance. In that case, the concrete *Publisher* would set the event source to be that object (as opposite to the above example).

Now, the concrete *Publisher* will fire the published events whenever such occurrences are detected. An event is lost if no concrete *Subscriber* registered interest in that event type. So, let's see how to code a concrete *Subscriber* and how to register interest in a published event. A concrete *Subscriber* has to implement the *Subscriber* interface. This interface has only one method, *eventFired*, which gets an *Event* as only argument. This method is a callback that notifies the event to the monitoring object. Here's an example:

```
package MyPkg::ConcreteSubscriber;

use base qw(OME::Util::PubSub::Subscriber);

      . . .

sub eventFired {
    my ($self, $event) = @_;
    # $event is a Event, you may need to cast…
    # OK, now do whatever needed
      . . .
}
```

At this point, you only need to register subscribers somewhere in your program, for example:

```
my $p = MyPkg::ConcretePublisher->new();
my $s1 = MyPkg::ConcreteSubscriber->new();
my $s2 = MyPkg::AnotherConcreteSubscriber->new();

# register $s1 for all published events
$p->register($s1);

# register $s2 only for ConcreteEvent_2
$p->register($s2,'MyPkg::ConcreteEvent_2');
```

Now, when the *ConcretePublisher* object fires *ConcreteEvent_2*, the *Publisher* dispatches that event instance to all the listeners (subscribers) in the corresponding event notification list (the set of subscribers that registered interest in that event type), which, in this case, happens to contain *s1* and *s2*. In contrast, if *ConcreteEvent_1* was fired, then only *s1* would be notified.

Naturally, subscribers can also dynamically be removed from any event notification list. This is done through the *remove* method of *Publisher*. So, let's conclude this section with the details of the *register* and *remove* methods that concrete publishers inherit from their abstract parent. These public methods are leaf methods, meaning that they're not polymorphic, that is, they may not be overridden in subclasses.

The *register* method:

```
$concretePublisher->register($subscriber [, $event1, $event2, ...]);
```

Adds a subscriber to the notification list of each specified event. If no event type is specified, then the subscriber will be added to all event lists. The subscriber will be notified of every occurrence of the specified event types - those have to be published events. No matter how many times a subscriber tries to register for an event, it will only be added once to the event notification list. This obviously means it will also be notified once for every occurrence of that event.

Arguments:
`$subscriber` an instance of *Subscriber*.
`$event1, $event2, ...` fully qualified names (packages) of event classes.

Return:
True if `$subscriber` has been added to the notification list of all specified events, false otherwise. False can be returned if one of the specified event types is not published or if `$subscriber` has already registered for any of the specified event types.

The *remove* method:

```
$concretePublisher->remove($subscriber [, $event1, $event2, ...]);
```

Removes a subscriber from the notification list of each specified event. If no event type is specified, then the subscriber will be removed from all event lists.

Arguments:
`$subscriber` an instance of *Subscriber*.
`$event1, $event2, ...` fully qualified names (packages) of event classes.

Return:
True if `$subscriber` has been removed from the notification list of all specified events, false otherwise. False can be returned if one of the specified event types is not published or if `$subscriber` wasn't registered for any of the specified event types.

References

[BRJ00] G. Booch, J. Rumbaugh, I. Jacobson:
The Unified Modeling Language User Guide
Addison-Wesley, 2000

[GoF95] E. Gamma, R. Helm, R. Johnson, J. Vlissides:
Design Patterns — Elements of Reusable Object-Oriented Software
Addison-Wesley, 1995

[OMG01] Object Management Group:
Unified Modeling Language Specification v1.4
Available from http://www.uml.org/

[POSA1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal:
Pattern-Oriented Software Architecture —A System of Patterns
John Wiley & Sons, 1996