Patterns for event notification

Jorge Mederos, Julio García

Universidad Politécnica de Madrid [1]

Abstract

This paper describes the NOTIFIER *and* INTEREST PUBLISHER *patterns for event notification in the context of graphical application design.The aim is to improve the known set of interaction patterns.The* NOTIFIER *pattern centralizes the event notification mechanics. The* INTEREST PUBLISHER *pattern describes the event representation and management. Both pattern can be combined to gain more flexibility.*

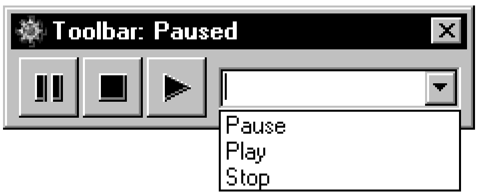
The notifier pattern

Intent

This template isolates and centralizes the many-to-many dependence between *n* objects changing their state and *m* objects depending upon them for a proper update.

Motivation

Graphical applications are not executed in a linear fashion from the beginning to the end. Instead, they are executed in an asynchronous fashion, waiting for user events like mouse clicks or keystrokes to perform their programmed actions. Graphic user interface objects such as buttons, lists or combo boxes, need to communicate with the rest of the application logic. To do this, they generate events of interest like *onButtonPressed*, *afterListItemInserted* or *onDropDown*.



The simple toolbar in the figure 1.1 shows the relationships between the user interface objects and the application logic. It has three buttons for video operations and an operation history list inside a combo box describing previous operations made by the user. When the user presses the play button, the button sends the *onButtonPressed* event to the application, and there are several actions to be performed: It will continue playing a video movie in another window and will update the operation list among others things, like changing the title bar for feedback when the drop down list is not dropped. The insertion of an operation in the history list will also generate the events of interest *beforeListItemInserted* and *afterListItemInserted*, which can produce further process by the application, as selecting the

[1] LSIIS Department, Facultad de informática, Campus de Montegancedo, Boadilla del Monte, 28660 Madrid, Spain. Email: jmederos@gran-via.com, juliog@fi.upm.es

correct localized language string for this operation. Application and user interface objects cooperate to bring the user the illusion of real videotape manipulation.

This communication is a problem itself. If the application code is tied directly to user interface events, then it makes the appropriate consecutive calls to other parts of the application needing to be notified: play a video movie, insert the correct language dependent operation string, and update the title bar to show the new operation (also in the adecuate language). The result is showed in figure 1.2. This kind of code tightly couples different application modules and objects with the application user interface control logic, not allowing further changes or improvements in an easy way.

Figure 1.2. *Toolbar state after pressing the play button with the operation history list dropped down.*

The problem arising now is how to synchonize these cooperative objects while maintaining them decoupled. Some known patterns deal with this problem. The OBSERVER pattern [Gam95; 293-303] also called PUBLISHER-SUBSCRIBER in [Coa95; 443] describes a decoupled solution for it, but it does not specify how to select specific events. The COURIER [Hel96] and CONDUIT OF SUBJECT CHANGE [Vli96; 76-86] patterns are variations of the previous patterns that take into account this issue. They are reviewed in the *Related patterns* section.

The NOTIFIER pattern is a special kind of a MEDIATOR pattern also found in [Gam95; 273-282]. Figure 1.3 shows the behaviour of this pattern.

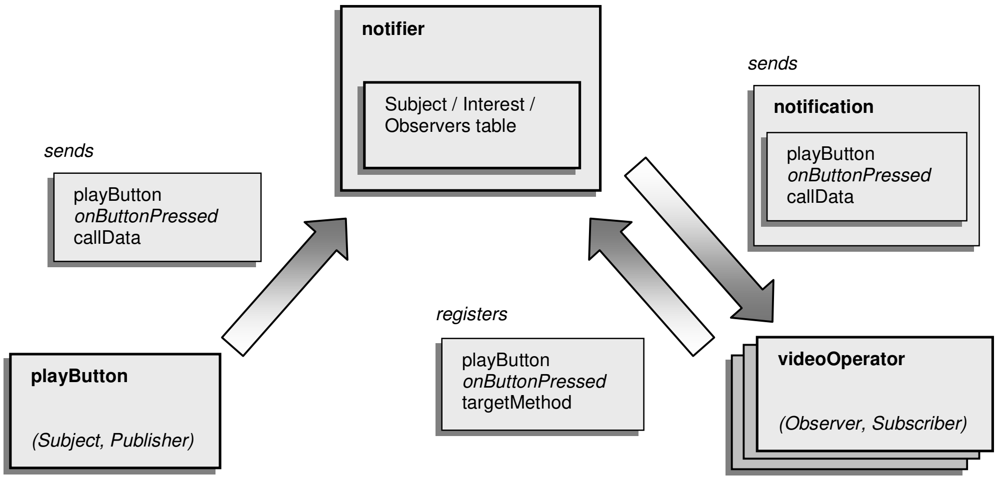


Figure 1.3. *NOTIFIER behaviour when the play button is pressed by the user.*

A **notifier** object mediates between the **playButton** and its observers, for example a **videoOperator** object. The **playButton** can send an ***onButtonPressed*** event and maybe others, like *onButtonDown* or *onButtonUp*. The **videoOperator** can choose some, in this example *onButtonPressed*, and subscribe to them with the **notifier**. When the user presses the play button on the toolbar, the state of **playButton** changes. It then chooses the appropriate event for this moment, like *onButtonPressed*, and notifies its change to the **notifier**. The **playButton** also sends additional **callData**, as for example the clock time of the event. The **notifier** maintains the list of dependent observers on the **playButton** object among others subjects. It notifies the **videoOperator** and other observers by calling their target methods, defined when they subscribed themselves with the **notifier**.

Applicability

The NOTIFIER pattern can be used in any of the following situations:

- When it is needed a more decoupled notifier subsystem than the offered by the OBSERVER, COURIER or CONDUIT OF SUBJECT CHANGE patterns.
- When it is important to maintain a centralized acquaintance of the *m-n* relationships established between the subjects and the observers. It may be necessary to be aware of them.
- When the behaviour of the notifier subsystem cannot be predicted, and it could be extended in the future, like supporting remote notifications or batched notifications.

Structure

Figure 1.4 shows the NOTIFIER structure. A **notifier** object mediates between its clients: subjects and observers. Subjects declare events of interest by using **interest** objects, and observers can choose and subscribe to some of them via the **notifier**. When the state of the subject changes, the subject chooses the appropriate event of interest and calls the **notifier**, notifying its change. The **notifier**, who maintains the list of dependent observers on that subject, then notifies the observers by calling their target methods, defined when they subscribed to the notifier. The information is held by a **notification** object.

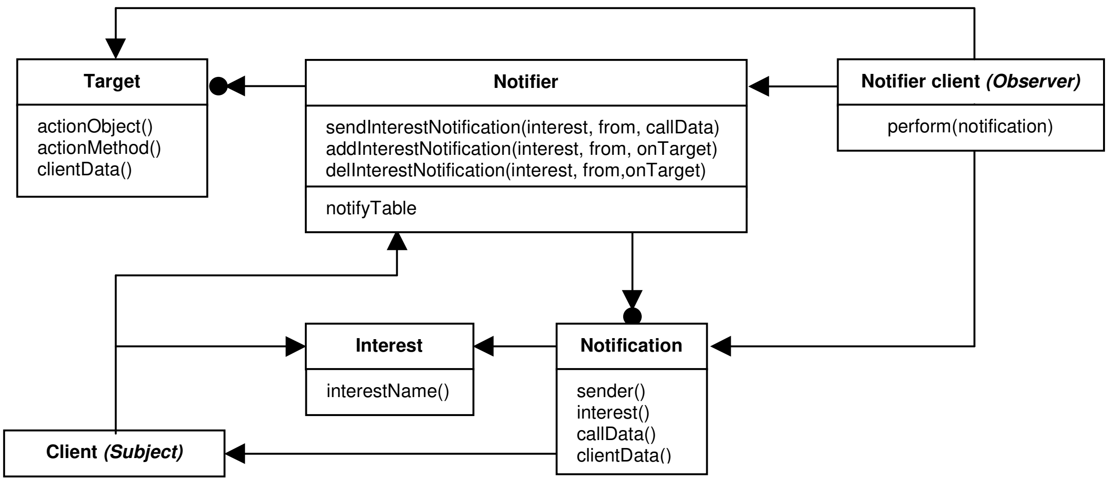


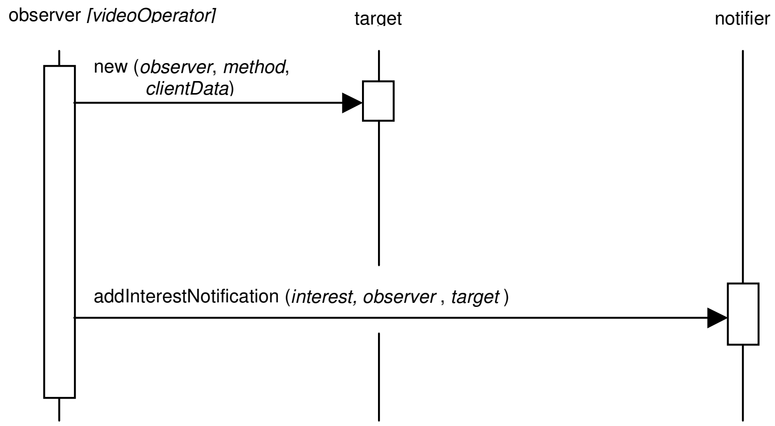
Figure 1.4. *Structure of the NOTIFIER pattern.*

Participants

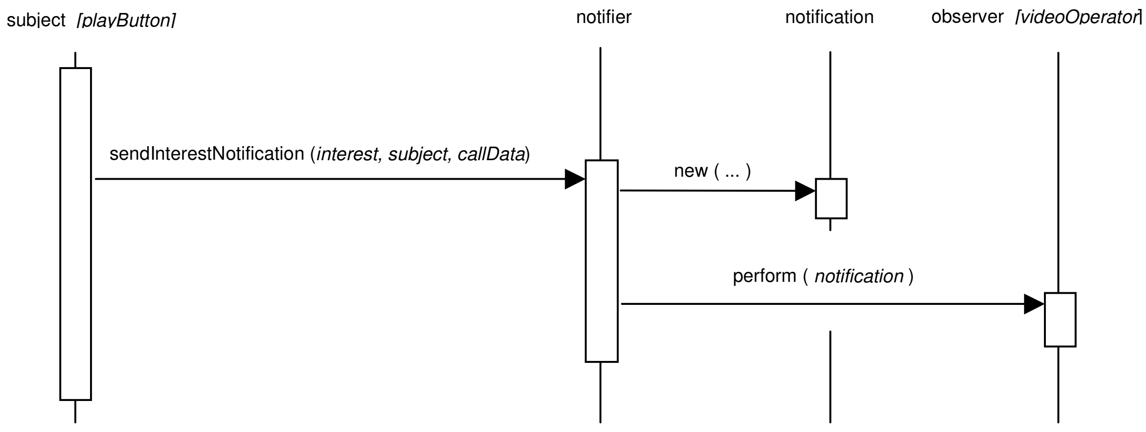
- **Client** (*Subject)*. Notifies state changes to observers by using a notifier.
- **Notifier client (***Observer*). Receives notifications of subjects via the notifier.
- **Interest.** Identifies an state change of a subject.
- **Notifier.** Stores which clients wish to send and receive event notifications. It uses interests to identify events, and targets to make calls to observers.
- **Notification.** Holds the notification information of the subject and the observer.
- **Target.** Defines a generic target call for any object.

Collaborations

- An observer requests to receive an interest notification from a subject.



- A subject sends an interest notification, allowing observers to know about its recent state change.



Consequences

- *Emphasizes the use of centralized communications*. They are simpler to work with and to maintain. It is not necessary to have only one notifier object. It is possible to assign a notifier to a related group of classes or objects, allowing to distribute the notifier load.

- *Abstracts coupling between subject and observer*. Like the OBSERVER, COURIER and CONDUIT OF SUBJECT CHANGE patterns. Relying on a notification subsystem decouples the subject and observer. The concrete classes of subjects and observers do not need to be known.
- *Makes easier the introduction of behaviour changes in the notification subsystem.* The notifier class can be derived to support other styles of notification, like batched notifications, asynchronous notifications or remote notifications. These new styles are not obstrusive with subjects and observers.
- *The notifier object needs to be known by its clients*. The subject and its observers use the notifier object as a communication channel. The notifier usually is shared by several subjects and observers. This is a serious constraint because the notifier has to be visible to its clients. Using a notifier attribute in subjects and observers exposes the notification mechanism, and ties it to a fixed design.
- *The table of the notifier can be huge if there are few of them.* This behaviour can degrade the overall system perfomance. The use of hash tables is a must. It is sometimes possible to share a lot of entries, because observers of a known class normally subscribe to the same interests on the same subject classes. If the table continues being huge, more notifiers are needed.
- *Many simultaneous notifications can overload the notification subsystem.* When multiple notifications are generated, it could be a serious problem how to handle them. Synchronous notifications could take a lot of time to complete. Therefore sophisticated notification schemes are needed. Batched notifications permit grouping of related notifications. Asynchronous notifications can be used without waiting for the observers to end their processing.

Related patterns

The OBSERVER pattern, also called PUBLISHER-SUBSCRIBER, describes a decoupled solution for it. The user interface objects, like the buttons and the drop down list, behave as subjects or publishers for some interface events. The **videoOperator** object behaves as subject or subscriber, registering to be notified by the play button. However, the pattern does not specify a way to select specific events to subscribe to.

The COURIER and CONDUIT OF SUBJECT CHANGE patterns are variations of the previous patterns taking into account this issue. They describe an intermediate object, a message, carrying the event type generated by the subject to the observers (and possibly extra information, like a string value for a *beforeListItemInserted* event).

All those patterns use an object list mantained by the subject to describe the 1-n relationship between itself and its observers. In the previous example, there are several objects from the user interface and the application implicated in a simple toolbar. Distributing the publish-subscript behaviour among all of them complicates further modifications. This situation can be simplified by using the NOTIFIER pattern.

Viewed as a whole thing, the relationship between n subjects and m observers can be described as a many-to-many relationship. A subject can have many observers interested in it and an observer can be interested in many subjects. Subjects can also behave as observers for other events of interest and vice versa. For example, the **toolbarTitle** object behaves as a subject if the title string changes, generating the events of interest *beforeItemChanged* and *afterItemChanged* to be maybe handled by other objects. The toolbar title also behaves as an observer of the event of interest *onButtonPressed* generated by the three buttons, knowing when it must change its label.

The INTEREST PUBLISHER pattern, also presented here, describes a more precise way to handle the events of interest by the subjects.

The SINGLETON pattern may be used for dedicated notifiers to make them unique.

Implementation

Benefits of object composition are more oportunities for object collaborations, maintaining its independence. When implementing these patterns, there are a few issues to revise:

1. *Using the **perform()** method to receive notifications or the **actionMethod()** of a **Target***. The notifier needs a known method in the observer to perform the notification call. There is an inherent limitation in many object-oriented languages when we are trying to know a method pointer. Sometimes it is impossible. The **perform()** method will be used in these cases. Then, clients inherit from a base **NotifierClient** class obtaining the **perform()** method. In C++ the notifier makes a simple method call with an object of the base **NotifierClient** class.

   ```
   notifierClient->perform(notification);
   ```

   Using the **actionMethod()** requires some degree of dynamic knoweledge of method pointers. In Objective-C or Smalltalk, for example, method invocation always is performed using *message passing*. A message consist in a string or numeric selector —its name— and its arguments. In Objective-C sending a message to an object is written enclosed in square brakets. Often, the return value of a message is an object too, so it can receive another message as well. For example:

   ```
   [anObject myMessageWith: aParameter andAnotherParameter: anotherParameter];
   [[anObject myMessage: aParameter] otherMessageWithoutParameters];
   ```

   When an object receives a message, it asks to its own class if it has a method responding that message. If it is true, the method is executed; if it is not, the message is sent to its superclass, doing the same check. Due the actual pointer of the method is not known until runtime, **Target** objects can store the selector for the message to be sent with the object who can respond to it.

   There is a message known by all objects, used to send a message with one parameter to any object: it is **perform:with:**. The message to be sent is identified by a selector. A sentence in Objective-C to send a message notification to any object that maybe has a method responding the **myMessage** selector is:

   ```
   [anObject perform: @selector("myMessage") with: aNotification];
   ```

   Note that the **Target** object is created by the observer, and to reffer to its target method implementation, it use the selector associed with that method, so it always will respond to the notification. Using this language feature, the **perform()** method is not neccessary. And then, there is no reason to inherit from a base **NotifierClient** class. So in this pattern there are two implementation choices to perform event notifications, depending on the language used.

2. *Sharing interests among clients*. A benefit of composition is a simple way to object sharing: object references. Objects tends to use the same interests if they belong to the same class,

so there is a reason to provide some degree of event of interest sharing. A good way to isolate sharing schemes is the use of an **InterestFactory** class, based on the ABSTRACT FACTORY pattern [Gam95; 87-95], and implementing the **Interest** class following the SINGLETON pattern [Gam95; 127-134].

3. *Distributing event notifications.* Using an external notifier simplifies the support for remote notifications. The remote notification logic should be confined in the **Notifier** class and some of its specialized descendants. A PROXY pattern [Gam95; 207-217], [Vli96; 105-118] can be used to handle the remote presence of all objects. Stand alone clients, as CORBA, DCOM, RMI or PDO objects can follow this pattern for client/server sinchronization, due its independence and protocol support.

The sample code implementations is written in Objective-C. In Objective-C an **id** means any class type. Recall to point 1 of this section to know a little more about how Objective-C makes calls to methods using messages. The **Notifier** class is declared as follows:

```
@interface Notifier : Object
{
    notifyTable;
}
+ new;
- free;
- sendInterestNotification: anInterest from: anObject withCallData: callData;
- addInterestNotification:  anInterest from: anObject onTarget: aTarget;
- delInterestNotification:  anInterest from: anObject onTarget: aTarget;
@end
```

An observer, like the **videoOperator,** constructs a target object pointing to one of its methods, and use the **addInterestNotification:from:onTarget:** message to receive notifications on the desired target. To ilustrate the use of a **clientData** object, maybe the **videoOperator** object works with some instances of the video toolbar. Imagine we want to know in which order the toolbar instances were launched. We can store the instance number in the **clientData**. The **clientData** represents the context of the observer for the interest requested.

```
id anInterest = [interestFactory createInterestNamed: "onButtonPressed"];
id vidToolbar = [new VideoToolbar];
id aTarget    = [new Target: self
                     method: @selector("playVideo")
                     clientData: [self incrVideoToolbarInstances]];

[[self notifier] addInterestNotification: anInterest
                 from: videoToolbar
                 onTarget: aTarget];
```

As an example of point 2 of this section, the **InterestFactory** class is used here to share the interest objects by all clients.

Subjects, like the **playButton**, use the **sendInterestNotification:from:withCallData:** message to expose state changes. The **callData** is used to send context information of the state change context of the subject, i.e. holds information specific to the current interest. For example, in a list, an *onSelectedItemChange* event of interest is generated whatever the user choose a different item in the list. A **callData** for this interest would be the row currently selected, the selected item string and the current time. In a **playButton**, *onButtonPressed* interest would only sends the current time.

```
Id anInterest = [interestFactory createInterestNamed: "onButtonPressed"];
```

```
[[self notifier] sendInterestNotification: anInterest
                 from: self
                 withCallData: [systemInfo currentTime]];
```

The **Notifier** class propagates notifications when it receives the **sendInterestNotification: from:withCallData** message. The notifier builds the notification object and calls the client object using the Objective-C predefined **perform:with:** message in the **Object** class, with the notification object as parameter. This message is reviewed in the point 1 of this section.

```
- sendInterestNotification: anInterest from: aSubject withCallData: callData
{
    id anItem;
    id aNotification;
    id aTarget;

    anItem = [[self notifyTable] itemMatching: anInterest from: aSubject];
    if ( anItem ) {
        aTarget = [anItem target];
        aNotification = [Notification new: anInterest
                                      withSender: aSubject
                                      withCallData: callData
                                      withClientData: [aTarget clientData]];
        [[aTarget actionObject] perform: [aTarget actionMethod]
                                with: aNotification];
        [aNotification free];
    }
}
```

Known uses

Some variations of the NOTIFIER pattern are used in the RPC service implementation and in the DCE RPC implementation. The exact server for a requested interface by an application client is queried to a well known RPC server provider. When the RPC servers go up, they register themselves in the RPC server provider, behaving as a subject. The RPC server provider is the notifier and the application client the observer. CORBA implementations use similar notifiers to connect CORBA servers to ORBs. For example, Orbix implementation [Ion95; 7-8] uses an Implementation Repository which maintains a mapping from a server's name to the file name of the executable code which implements that server. CORBA servers must registers themselves in it behaving again as subjects. CORBA clients behave as observers.

The CommonPoint framework uses a notification framework for event notification [Coo95; 328-336]. There is a **Notifier** class, an **Interest** class for event description and a **Model** class to communicate with the notification framework.

There is also a notifier subsystem in most graphical user interfaces implementations, behaving as a NOTIFIER who dispatches the events from subjects —graphic objects— to observers —graphic applications—. For example, the notifier in the X Window System is the X Server [Hel93; 3-9], and the XView user interface uses a notifier object to centralize all interface and operative system event notifications [Hel93; 459-509].

The interest publisher pattern

Intent

Publishes state changes or events of interest of an object, allowing to share conceptually similar events with other objects.

Motivation

Often objects need to cooperate. Graphical applications are a good example of objects collaboration in action. All graphical user interfaces have graphical objects that generate events of interest to describe what is happening. Other objects can receive these events and decide to do something. As a simple example, a video toolbar has several objects, like buttons to play, pause or stop a video movie. A **videoOperator** object, part of a graphical video viewer application, can wait until one of the buttons in the toolbar is pressed.

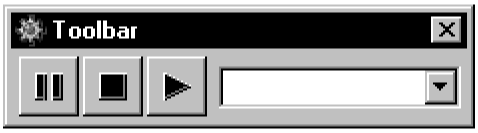


Figure 2.1. Simple video toolbar for playing video movies. The buttons in the toolbar generates events for a video viewer application

In this situation, a button generates an *onButtonPress* event of interest and the **videoOperator** decides to play, pause or stop the movie accordly with the button meaning. There are several other events in an application. For example, a timer object may generate *newAlarm* events. And a networked application may generate a *newConnection* event.

Graphical user interface frameworks always define a set of events that can be generated by each user interface object. But often there is no way to know what this set is. Custom application events, if they are defined, usually have the same problem: the events are handled implicitly in the design. An application would not be interested in low level events as *onButtonPress* or *newAlarm*. Instead it would like to define events as *playVideo* or *timeToRecord*. Moreover, to gain simplicity, an application would may want to use common higher level events and hide the rest. For example, a *newAlarm* event for playing the video movie would be translated to a *timeToPlay* event. But maybe, it would be better generate again the *playVideo* event, because lesser event of interest to remember leds to simpler event relationships between objects. Thus, different threads of executions like a timer or a play button click, can easyly produce the same results if, for example, the **videoOperator** only plays a video movie when a *playVideo* event of interest is generated.

Reffering to an event of interest would be another problem. Often, they are identified by simple integer constants, character strings or both, not existing as a separate entity, which leds to inconsistent management of them. Worst, the application would be using a different representation of events than the user interface or other application libraries. It should be better to have one *event of interest* entity defined. It allows meaningful comparisons between all of them, hidding the implementation details.

The pattern describes a way to organize these issues. An INTEREST PUBLISHER publishes state changes or events of interest of an object, allowing to share conceptually similar events like *playVideo* between objects, and maintaining a common reffering way to them. The INTEREST PUBLISHER identifies each potentially public state change with an interest object. When an object needs to know which are the public events of interest of another publisher, it asks him for a list of them. This pattern can be found implicit in many interaction patterns, as reviewed in the *Related patterns* section.

Aplicability

Use the INTEREST PUBLISHER pattern in any of these situations:

- When an object needs to publish events of interest, but it is important to maintain a common access to them across the application, without worrying about the exact interface of each object. Interest objects are used for this purpose.
- When it can be predicted that certain common state changes will be shared by different classes. Common interest objects can be used to describe them. The *playVideo* event of interest would be one of them.
- When objects may want to know what are the events of interest another object can raise, to select some of them to subscribe. The query interest interface offers this fuctionality.

Structure

The INTEREST PUBLISHER mantains a list of event of interest to be queried by other clients behaving as observers. Concrete publishers who are subjects of interest of other objects publish the events of interest it will raise.

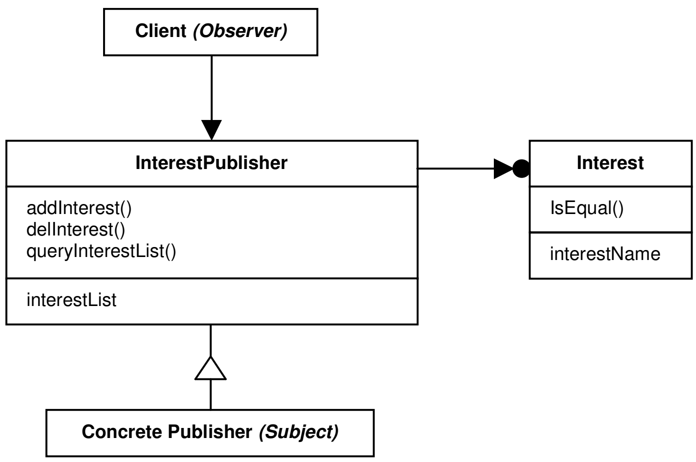


Figure 2.2. *Structure of an INTEREST PUBLISHER pattern.*

Participants

- **Client.** (*Observer*). Use the interst publisher query interface to know about event of interest published by subjects.

- **Interest Publisher**. Mantains an interest lists for its clients.
- **Concrete Publisher.** Publish event of interest using the interest publisher. Common concrete publishers are other interaction patterns.
- **Interest.** Identifies an state change of an object published by the subject.

Collaborations

- A concrete publisher uses the inherited **addInterest()** to publish an internal state change, and **delInterest()** to remove a previous published interest.
- A client use **queryInterestList()** to know about the publishied interests of a subject.

Consequences

- *The interests allow cooperation of classes using a common state change language.* The state change languaje based on events of interest descriptions isolates existent incompatible implementation-dependent mechanisms for state change notification. Each object can use whatever algorithm it want to change internally its state. Then, if it is appropiate, choose a published interest to raise.
- *Semantically similar states can be represented with the same interest.* A common interest languaje can be used to simplify the relations between objects trying to group similar internal state changes of different objects with a common interest.
- *An interest publisher inherently needs a notification mechanism.* Only publishing a set of interests is not useful enought. The subject needs to communicate the interests to other objects. Inherited classes can implement internally an OBSERVER pattern for simple notifications, but for complex ones a COURIER, CONDUIT OF SUBJECT CHANGE or NOTIFIER pattern are more appropiate.

Related patterns

The representation of events to subscribe and how to handle them is not defined in the interaction patterns. But they tends to use a kind of event representation. A blur description leads to multiple incompatible representations.

The OBSERVER or PUBLISHER-SUBSCRIBER, COURIER, CONDUIT OF SUBJECT CHANGE and NOTIFIER patterns can use the INTEREST PUBLISHER pattern to define a way to handle a common set of event of interest. They may act as from it to gain the interest publishing functionality. Thus, they may be considerated as variants of previous patterns.

Implementation

1. The **InterestPublisher** class is quite strainforward. It declares an **interestList** and the basic functions to deal with it. The declaration in Objective-C is the following. The **id** type stands for any class type:

```
@interface InterestPublisher : Object
{
    id interestList;
```

```
}
+ new;
- free;
- addInterest: anInterest;
- delInterest: anInterest;
- queryInterestList: anInterest;
@end
```

2. The **Interest** class has to define the representation of an event of interest. Maybe it can define more than one representation, like a numeric and a string representation. The **Interest** class also can unify the user interface and application events providing an unique interface to both. This property can help in the design of frameworks for multiple graphical user interfaces.
3. Instances of an **Interest** class tends to be inmutables. The **Interest** class may use the SINGLETON pattern to make each different instance unique. An ABSTRACT FACTORY pattern may hide the interests instance construction as well. However, this is not true in interprocess communications, because the address space changes. Interests are lightweight objects and is probably better to make copy of them in a remote query by a client.
4. Subjects belonging to a same class usually publish the same interests. For example, a graphical **Button** class may publish the interests named *onButtonPressed* and *onButtonReleased*. If all instances of **Button** class will use the same interests, you can choose not to inherit from an **InterestPublisher** class, but delegate it in an instance of this class shared between all **Button** instances.

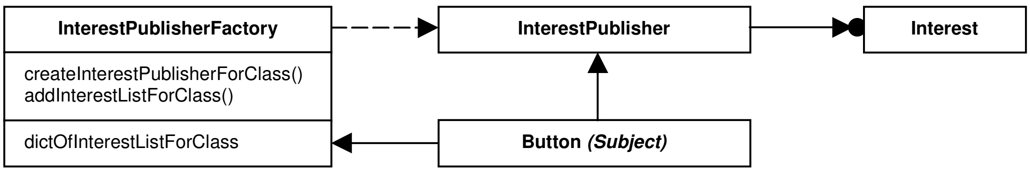


Figure 2.3. *Implementation of an INTEREST PUBLISHER pattern using delegation. The **ConcretePublisher** delegates in a shared **InterestPublisher** (using the SINGLETON pattern) the Interest management.*

To simplify the construction of shared **InterestPublisher** instances depending on class types, we can use an **InterestPublisherFactory**. It declares a dictionary variable to store interest lists of each **InterestPublisher** instance. The key value is the class of the subject, in this example the **Button** class. Note that in Objective-C, like in Smalltalk, the classes are objects too. The **InterestPublisherFactory** class declaration is the following:

```
@interface InterestPublisherFactory : Object
{
    id dictOfInterestListForClass;
}
+ new;
- free;
- addInterestList: interestList forClass: aClass;
- createInterestPublisherForClass: aClass;
@end
```

Foreach class we may share, we use the **addInterestList:ForClass:** message, registering the interests associed with each class.

```
id  interestList = [List new];
[interestList addItem: [Interest new: "onButtonPressed"]];
```

```
[interestList addItem: [Interest new: "onButtonReleased"]];

[interestPublisherFactory addInterestList: interestList
                          forClass: [Button class]];
```

The method for the **addInterestList:ForClass:** message is defined in **InterestPublisherFactory** as:

```
- addInterestList: interestList forClass: aClass
{
    [dictOfInterestListForClass addItem: interestList forKey: aClass];
    return self;
}
```

Later, when we use c**reateInterestPublisherForClass:** to create a dedicated **InterestPublisher** instance for the **Button** class, the **InterestPublisherFactory** adds the previously registered interest list to the instance of **InterestPublisher**.

```
- createInterestPublisherForClass: aClass
{
    id interestPublisher = [new InterestPublisher];
    id interestList      = [dictOfInterestListForClass itemForKey: aClass];

    if ( interestList ) {
        for ( i = 0; i < [interestList count]; i++ )
            [interestPublisher addInterest: [interestList at: i]];
    }
    return interestPublisher;
}
```

To create an InterestPublisher instance for the Button class we send the c**reateInterestPublisherForClass:** message to the factory object:

```
[interestPublisherFactory creteInterestPublisherForClass: [Button class]];
```

5. The INTEREST PUBLISHER pattern would usually used with other interaction patterns. As an example, we show how to do it with the NOTIFIER pattern.

   We have to define a concrete subject, as for example **NotifierButtonPublisher**, to deal with the **Notifier** class. The NOTIFIER pattern uses a simple **Interest** class. We can sustitute this class with the **InterestPublisher** and **Interest** classes described in the INTEREST PUBLISHER pattern.

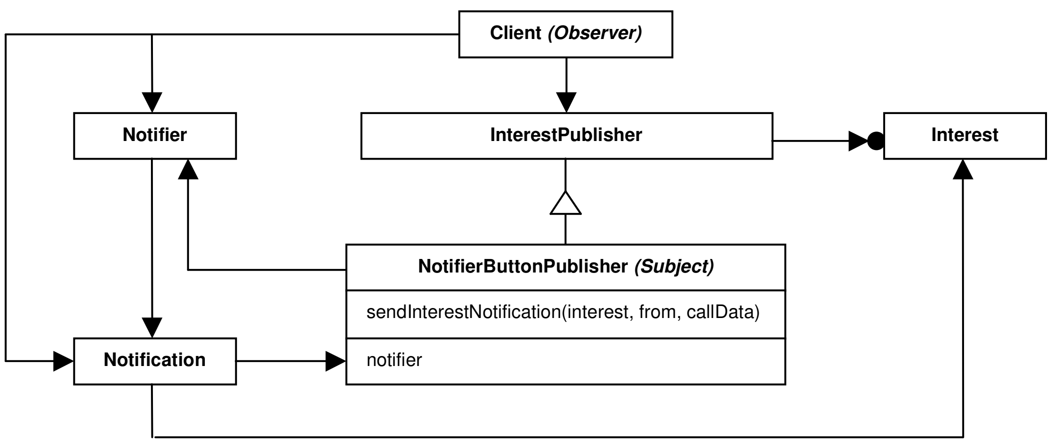


Figure 2.4. *Use of an INTEREST PUBLISHER pattern with a NOTIFIER pattern inheriting from **InterestPublisher** class.*

The **NotifierButtonPublisher** class is a concrete publisher inheriting from the **InterestPublisher**. Its declaration is the following:

```
@interface NotifierButtonPublisher : InterestPublisher
{
    id notifier;
}
+ new: aNotifier;
- initialize: aNotifier;
- free;
- sendInterestNotification: anInterest withCallData: callData;
@end
```

The constructor **new:** calls its superclass to allocate space and initializes it sending the **initialize:** message to the new object allocated, setting the interests to be published. **self** stands for the object instance (like **this** in C++) and **super** stands for the parent class. The '**+**' represents a class message, the '**-**' indicates an instance message. For this reason the **new** message is sent to the superclass object (recall that in Objective-C classes are objects too) . The **initialize** is an instance message, so it is sended to **self**, which is the actual object instance.

```
+ new: aNotifier
{
    return [[super new] initialize: aNotifier ];
}

- initialize: aNotifier;
{
    notifier = aNotifier;
    [[self addInterest] addInterest: [Interest new: "onButtonPressed"]];
    [[self addInterest] addInterest: [Interest new: "onButtonReleased"]];
}
```

The **sendInterestNotification:withCallData:** message forwards the message to the notifier, sending the notification to the observers.

```
+ sendInterestNotification: anInterestName withCallData: callData
{
    [notifier sendInterestNotification: [self queryInterestList: anInterestName]
              from: self
              withCallData: callData]];
    return self;
}
```

The user interface code that handles the implementation-dependent *onButtonPressed* event, can send a notification using **sendInterestNotification:withCallData:**, passing the current time as call data parameter.

```
[notifierButtonPublisher sendInterestNotification: "onButtonPressed"
                         withCallData: [systemInfo currentTime]];
```

Known uses

The CommonPoint framework uses an **Interest** class for event description and a **Model** class to communicate with the notification framework. This class mantains the list of events of interest trigged by the data models derived from the **Model** class.

The INTEREST PUBLISHER pattern also can be found implicit in most graphical user interfaces implementations. They use a standard collection of event types and their descriptions.

Graphical applications use this common event languaje to communicate with user interface objects. Graphical objects define a set of events to trigger, as in the INTEREST PUBLISHER.

Bibliography

[Coa95] Peter Coad, David North, Mark Mayfield. *Object models: strategies, paterns, and applications*. Prentice Hall, Englewood Cliffs, NJ, 1995.

[Coo95] Sean Cooter. *Inside Taligent Technology*. Addison-Wesley, Reading, MA, 1995.

[Gam95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns. Elements of reusable object-oriented software*. Addison-Wesley, Reading, MA, 1995.

[Hel93] Dan Heller, Thomas Van Raalte. *XView Programming Manual for XView Version 3.2*. O'Reilly & Associates, Inc. 1993.

[Hel96] Richard Helm, Erich Gamma. *The Courier Patern*. Dr. Dobb's Sourcebook. january/february 1996.

[Ion95] *Orbix Programmer's Guide. Release 1.3*. Iona Technologies Ltd. July 1995.

[Vli96] *Pattern Languajes of Program Design 2*. Edited by John Vlissides, James Coplien and Norman Kerth. Addison-Wesley, Reading, MA, 1996.