
Extending OMG standard services

What is the recommended approach ?

Juan J. Hierro (Telefónica I+D), Fong Shen (Hewlett-Packard)
jhierro@tid.es, fshen@cup.hp.com

21/11/97

1 Introduction

As noticed in the corresponding RFP, the Notification Service is “essentially an extension of the existing OMG Event Service specification” which defines standardized interfaces in order to support:

- filtering of events (notifications) based on their types and/or contents
- dynamic addition of filterable event (notification) types
- configuration of QoS parameters (assured delivery, priority of events, lifetime/aging of events, etc)

The RFP explicitly requests that the Notification Service be based on the OMG Event Services specification. Specifically, “the programming model of Notification Services should extend that of the previously defined OMG Event Service specification”. Unfortunately, nothing is said about how such extension should be implemented (or at least, what principles should be preserved in the extended programming model, e.g., whether programs using a `CosEventChannelAdmin::EventChannel` should work without changes or not).

Different approaches may be adopted to implement the concept of “extension”:

- making use of inheritance and narrowing
- making use of inheritance, adding operations to avoid narrowing
- making use of inheritance, preventing narrowing
- Revising the currently adopted (standard) interfaces

This paper describes and analyzes each of them.

HP and Telefónica I+D would like to ask two questions to the AB. First, we would like to know if they agree in that the last choice in the list is valid. Second, we would like to know what is the approach being recommended by the AB.

We would like to point out that this is the first time a service is going to be defined as an extension of a previous one. For consistency reasons, we think that the approach to be adopted in definition of the Notification Service should also be applied if extension of other OMG services is required in the future. Consequently, we believe that definition of a pattern for extending adopted OMG specifications would be more appropriate than leaving the question to be answered by a restricted group of companies (the current group of submitters to the Notification Service RFP).

2 The points where extension is going to be applied

In order to make description of the different approaches easier to understand, we will detail how each of the approaches may be applied to extend a subset of interfaces in the OMG Event Service specification:

- the `CosEventChannelAdmin::EventChannel` interface
- the `CosEventChannelAdmin::ConsumerAdmin` interface
- the `CosEventChannelAdmin::ProxyPushSupplier` interface

There is a wide consensus in that these three interfaces (plus others) will be extended in the final Notification Service specification. The `CosEventChannelAdmin::EventChannel` interface will be extended to support operations that enable to set QoS parameters at the channel, for example. The `CosEventChannelAdmin::ConsumerAdmin` interface will be extended to support operations that enable to set filters that affect a group of proxies, for example. The `CosEventChannelAdmin::ProxyPushSupplier` interface will be extended to support operations that enable to set filters on a per-proxy (i.e., per-consumer) basis.

The main differences between the different approaches described in following sections are related to the way references to `ConsumerAdmin` and `ProxyPushSupplier` objects are obtained. Each of the approaches implies a slightly different programming model, some of them closer to the OMG Event Service programming model than others.

3 Analysis of the different approaches

3.1 Making use of inheritance and narrowing

In this approach, a new module will be defined (let's name it as `CosNotificationChannelAdmin`). This module includes interfaces that extend the standard `CosEventChannelAdmin::EventChannel`, `CosEventChannelAdmin::ConsumerAdmin` and `CosEventChannelAdmin::ProxyPushSupplier` interfaces via inheritance. **Operations defined in the Event Service** (`for_consumers`, `for_suppliers` and `obtain_*`) **are reused** for creation of objects that support the extended interfaces. **No additional operations are required** for creation of these objects.

Definition 3-1

```
module CosNotificationChannelAdmin {
    ...
    interface ProxyPushSupplier : CosEventChannelAdmin:ProxyPushSupplier {
        void add_filter (...);
        ...
    };

    interface ConsumerAdmin : CosEventChannelAdmin::ConsumerAdmin {
        void add_filter (...);
        ...
    };

    interface NotificationChannel : CosEventChannelAdmin::EventChannel {
        ...
    };
};
```

Making use of inheritance and narrowing (interfaces definition)

This approach makes use of the following fundamental concepts of the CORBA model:

- a CORBA object may export an interface X, but only a limited set of operations defined in interface Y (X inherits from Y) may be visible through a CORBA object reference variable `ref`, in a given program
- in order to obtain a wider visibility, the programmer need to narrow variable `ref` (this may imply using casting or invoking specific narrow operations, depending on the language mapping)

Thus, to obtain a reference that points to an object exporting the `CosNotificationChannelAdmin::ConsumerAdmin` interface, we simply need to invoke the `for_consumers` operation inherited by all `CosNotificationChannelAdmin::NotificationChannel` objects. Only the `CosEventChannelAdmin::ConsumerAdmin` interface would be visible using the reference value returned by the `for_consumers` operation, but it is true that visibility of the rest of operations may not be required in all programs.

No extra operations to obtain ConsumerAdmin objects need to be defined. If a wider visibility is required at some given point, the previously obtained reference value may be narrowed. Here, **the concept of narrowing is used to gain visibility** of operations exported by an object.

Analogously, to obtain a reference that points to an object exporting the `CosNotificationChannelAdmin::ProxyPushSupplier` interface, we simply need to invoke the `obtain_push_supplier` operation inherited by all `CosNotificationChannelAdmin::ConsumerAdmin` objects. Invoking that operation causes creation of a new `CosNotificationChannelAdmin::ProxyPushSupplier` object, but only the `CosEventChannelAdmin::ProxyPushSupplier` interface would be visible through the returned reference value. If a wider visibility is required at some given point, that reference value may be narrowed.

Example 3-1 illustrate how a program will obtain references to `ConsumerAdmin` objects and create references to proxies, as well as how visibility of operations may be gained in order to invoke operations to add filters, etc.

Example 3-1

```
CosNotificationChannelAdmin::NotificationChannel_ptr notif_channel;

// A reference to the default ConsumerAdmin object associated to
// the Notification channel is obtained by invoking for_consumers:

CosEventChannelAdmin::ConsumerAdmin_var consumer_admin =
    notif_channel -> for_consumers ();

...

// Later, a push consumer may require connection to the channel
// in order to obtain events that pass some given filter. This would
// involve two steps:

// in the first step, a connection is established:

CosEventChannelAdmin::ProxyPushSupplier_var my_proxy =
    consumer_admin -> obtain_push_supplier ();

// in the second step, visibility of operations that support definition
// of filters is gained using narrowing, and the filter is set:

CosNotificationChannelAdmin::ProxyPushSupplier_var my_notif_proxy =
    CosNotificationChannelAdmin::ProxyPushSupplier::_narrow (my_proxy);
my_notif_proxy -> add_filter (...);

...

// At some given point, we may decide to set a filter which affects
// the whole set of consumers connected to the channel. That action
// would require getting visibility of operations to set filters
// exported by ConsumerAdmin objects:

CosNotificationChannelAdmin::ConsumerAdmin_var notif_consumer_admin =
    CosNotificationChannelAdmin::ConsumerAdmin::_narrow (consumer_admin);

notif_consumer_admin -> add_filter (...);
```

Making use of inheritance and narrowing (usage example)

HP and Telefónica I+D think that this approach is elegant, reuses as much as possible from the existing OMG Event Service without adding too much extra operations, and the resulting interfaces are cleaner (operations inherited from the OMG Event Service don't need to be replicated in Notification Service interfaces).

The resulting programming model is rather close to the one defined in the OMG Event Service. A program based on the Event Service can be easily modified to, let's say, restrict the number of events received by a consumer: you will only need to narrow the reference of the proxy supplier connected to that consumer and then add filters using operations exported by the narrowed proxy.

3.2 Making use of inheritance, adding operations to avoid narrowing

In this approach, a new module will be defined (let's name it also as `CosNotificationChannelAdmin`). This module includes interfaces that extend the standard `CosEventChannelAdmin::EventChannel`, `CosEventChannelAdmin::ConsumerAdmin` and `CosEventChannelAdmin::ProxyPushSupplier` interfaces via inheritance. **Operations defined in the Event Service** (`for_consumers`, `for_suppliers` and `obtain_*`) **may be used** for creation of objects that support the extended interfaces. **However, their usage is deprecated because new operations are defined which essentially make the same thing but return reference values of the extended interface types.** Using these new operations **the programmer doesn't need to perform narrowing.**

Definition 3-2

```
module CosNotificationChannelAdmin {
    ...
    interface ProxyPushSupplier : CosEventChannelAdmin::ProxyPushSupplier {
        void add_filter (...);
        ...
    };

    interface ConsumerAdmin : CosEventChannelAdmin::ConsumerAdmin {
        void add_filter (...);
        ...
        ProxyPushSupplier obtain_notification_push_supplier ();
    };

    interface NotificationChannel : CosEventChannelAdmin::EventChannel {
        ...
        ConsumerAdmin for_notification_consumers ();
        SupplierAdmin for_notification_suppliers ();
    };
};
```

Making use of inheritance, adding operations to avoid narrowing (interfaces definition)

Use of narrowing, as described in Section 3.1, is allowed in this approach. However, it is deprecated and usage of specific operations is recommended instead.

Example 3-2 illustrate how a program will obtain references to `ConsumerAdmin` objects and create references to proxies.

Example 3-2

```
CosNotificationChannelAdmin::NotificationChannel_ptr notif_channel;

// A reference to the default ConsumerAdmin object associated to
// the Notification channel is obtained as follows:

CosEventChannelAdmin::ConsumerAdmin_var consumer_admin =
    notif_channel -> for_notification_consumers ();

...

// Later, a push consumer may require connection to the channel
// in order to obtain events that pass some given filter:

CosEventChannelAdmin::ProxyPushSupplier_var my_notif_proxy =
    consumer_admin -> obtain_notification_push_supplier ();

my_notif_proxy -> add_filter (...);

// At some given point, we may decide to set a filter which affects
// the whole set of consumers connected to the channel:

notif_consumer_admin -> add_filter (...);
```

Making use of inheritance, adding operations to avoid narrowing (usage example)

The arguments used by supporters of this approach are basically two:

- programmers don't need to include calls in order to narrow references (this argument is valid for programming languages which explicitly support narrow operations like C++)
- narrowing have a significant cost

We believe that the first argument is just a matter of taste and there is people that wouldn't consider it so relevant as to make a decision.

The second argument requires a deeper analysis. It is based on the assumption that narrowing may require a remote invocation to the target object (an `_is_a` operation) when IIOP is used, thus implying some cost.

The first thing we may say in response to this argument is that clients of a NotificationChannel will typically connect once at the beginning of their execution and, after that, will supply/consume events for a long time. Narrowing will be required only at the beginning of a client execution. In the event that narrowing implies a remote invocation, the cost of that invocation would be insignificant compared to the overall processing time of the client.

The second thing we may say is that narrowing basically requires a remote invocation if the reference being narrowed didn't contain type information. This happens if the reference was marshalled at the server with a Null type ID. Here, it is worth making a reference to the text of the revised IIOP specification:

“The type ID (in IORs) is provided to allow ORBs to preserve strong typing. This identifier is agreed on within the bridge and, for reasons outside the scope of this interoperability specification, needs to have a much broader scope to address various problems in

system evolution and maintenance. Type IDs support detection of type equivalence, and in conjunction with an Interface Repository, allow processes to reason about the relationship of the type of the object referred to and any other type.

The type ID, if provided by the server, indicates the most derived type at the time the reference is generated. The object's actual most derived type may later change to a more derived type. Therefore, the type ID in the IOR can only be interpreted by the client as a hint that the object supports at least the indicated interface. The client can succeed in narrowing the reference to the indicated interface, or to one of its base interfaces, based solely on the type ID in the IOR, but must not fail to narrow the reference without consulting the object via the `_is_a` or `_get_interface` pseudo-operations."

A smart implementation of a NotificationChannel may include the most derived type of the admin and proxy objects in the type ID field of all references returned to remote clients. Then, non-naive ORB implementations at the client would use this type ID to find out that the narrowing is valid, without requiring any remote invocation. Our conclusion in this respect, is simple: the argument against narrowing fails if we consider environments where things are properly implemented.

Talking about drawbacks of this approach, the fact is that we are duplicating operations that essentially make the same thing just to prevent inserting narrowing calls in the source code. This doesn't seem to be a big deal, given the fact that narrowing is a valid concept, frequently used in CORBA environments (overall, when generic services are used). What should we do if we decide to extend the Notification Service and define a new service which extends the capabilities of admin a proxy objects ? Should we add a new operation to create this new type of admin or proxy objects, thus having three different operations (i.e., `for_consumers`, `for_notification_consumers` and `for_enhanced_notification_consumers`) ?

We believe that a good approach is validated when it can be recursively applied giving reasonable results. We will obtain an explosion in the number of operations if the approach proposed in this section is applied. Such effect doesn't exist if the approach described in Section 3.1 is applied.

Last but not least, it is worth noticing that even in this approach, the best way to modify existing programs, in order to make them use filtering, etc with the minimum amount of changes in source code, would still be the one based on usage of narrowing. Otherwise, the programmer would need to change the name of the type associated to most of the variables as well as the name of operations used in his program: a task which, at least, will require recompiling the whole program.

3.3 Making use of inheritance, preventing narrowing

In this approach, a new module will be defined (let's name it also as `CosNotificationChannelAdmin`). This module includes interfaces that extend the standard `CosEventChannelAdmin::EventChannel`, `CosEventChannelAdmin::ConsumerAdmin` and `CosEventChannelAdmin::ProxyPushSupplier` interfaces via inheritance. **Operations defined in the Event Service** (`for_consumers`, `for_suppliers` and `obtain_*`) **cannot be used** for creation of objects that support the extended interfaces. **New operations for creation of these type of objects must be used instead.**

Basically, this approach is similar to the one in previous section but prevents narrowing references returned by the `for_consumers`, `for_suppliers` and `obtain_*` operations. Consequently, the approach seems to have all the disadvantages described in previous section and loses all the benefits we gained if we preserve the ability to narrow returned references: existing programs which made use of the OMG Event Service wouldn't be able to evolve unless we edit them, change all the identifiers that is necessary and recompile the program.

3.4 Revising the currently adopted (standard) interfaces

An interesting approach that requires validation by the AB would consist in adopting the Notification Service as if it were a revision of the existing OMG Event Service.

In the same way as operations are added to the ORB interface or CORBA IR interfaces, during revision of the corresponding specifications, we may add operations for filtering, assignment of event priorities/lifetimes, etc into the existing `CosEventChannelAdmin::ConsumerAdmin` and `CosEventChannelAdmin::ProxyPushSupplier` interfaces, for example.

Note that with this approach, discussion regarding whether making use of narrowing represents the best solution or not, will be eliminated.

Adopting this approach may affect the way OMG specifications are contemplated: an OMG adopted specification would contain definitions of a set of interfaces, each of which comprises a given set of operations which is suitable of being extended in future versions of the specification.

However, this doesn't need to be a problem and even may be seen as a benefit.

4 Conclusions

Different approaches may be adopted to extend existing OMG specifications. We believe that it is worth discussing the pros and cons of these approaches and ask the AB for a recommendation.

When applied to the problem of extending the OMG Event Service, the questions we would like to have answered are the followings:

- May the Notification Service be defined as revision of current OMG Event Service (thus, producing a new version of the OMG Event Service in practice) ?
- Which approach, among those described in this paper, would be recommended in order to extend the OMG Event Service ?