The Event Notification Pattern—Integrating Implicit Invocation with Object-Orientation

Dirk Riehle
UBILAB, Union Bank of Switzerland
Bahnhofstrasse 45, CH-8021 Zürich
E-mail: riehle@ubilab.ubs.ch

Abstract

Managing inter-object dependencies in object-oriented systems is a complex task. Changes of one object often require dependent objects to change accordingly. Making every object explicitly inform every dependent object about its state changes intertwines object interfaces and implementations, thereby hampering system evolution and maintenance. These problems can be overcome by introducing the notion of Implicit Invocation to object-oriented systems as a decoupling mechanism between objects. This paper presents the Event Notification pattern, a pattern to smoothly integrate implicit invocation mechanisms with object-oriented designs. State changes of objects, dependencies of other objects on them and the maintenance links between these objects are made explicit as first class objects. The resulting structure is highly flexible and can be used to manage inter-object dependencies in object-oriented systems efficiently.

1 Introduction

Object-orientation has been acknowledged as a major architectural style, that is an organizational pattern for software systems. Numerous projects have shown that this style scales well from the most basic to very large systems. In practice, however, it is seldom found in its pure form: most object-oriented systems enhance the basic paradigm with other styles to solve specific problems.

One of these problems is the maintenance of update dependencies between objects. Changes to one object may affect other objects which subsequently have to be updated. This paper shows how this dependency can be established and maintained without intertwining object interfaces and thus hampering system evolution. This is done by introducing the architectural style of Implicit Invocation to object-oriented systems.

In a system based on Implicit Invocation, components (or objects) do not know each other explicitly. Rather, they communicate by announcing events. Objects register for events, so that announcing an event by one object leads to the notification of those objects which have registered for the event. From the changed object's perspective, the invocation of dependent objects' operations is implicit, which gave the style its name. This decoupling mechanism can be used to manage inter-object dependencies in object-oriented systems efficiently.

This paper draws on previous work on Implicit Invocation as well as our own experiences (Sullivan & Notkin, 1992; Riehle & Züllighoven, 1995). It presents an integration rationale for Implicit Invocation with object-oriented systems and elaborates it as the Event Notification pattern. This pattern is conceptually similar to the Observer pattern (Gamma et al., 1995). However, it has a different structure and lends itself to different implementations that overcome some of the disadvantages of the Observer pattern. In particular, the pattern presents an implementation technique based on parameterized types (templates in C++) which achieve type-safe forwarding of event notification parameters thereby overcoming problems of earlier implementations of implicit invocation mechanisms.

Both the Event Notification and the Observer pattern can be used in the design and implementation of a wide ranging number of software systems, for example interactive software systems, software development environments, distributed systems and real-time monitoring systems. Each of these systems benefits from the integration, sometimes in different respects. This paper focusses on the decoupling of object interfaces to ease system evolution and maintenance.

Section 2 of this paper discusses the Implicit Invocation style and why it is introduced to object-oriented systems. It further presents the rationale of the Event Notification pattern and shows in which respects it differs from the Observer pattern. Section 3 presents the actual Event Notification pattern. Section 4 discusses related work on Implicit Invocation and section 5 summarizes the paper and presents some conclusions.

To be published in *Theory and Practice of Object Systems* 2, 1 (1996).

2 Implicit Invocation in Object-Oriented Systems

Implicit Invocation has been defined as an architectural style in (Garlan & Shaw, 1993; Shaw, 1995). A system based on implicit invocation consists of several components with a granularity ranging from simple objects to full-fledged reactive processes. They are connected through implicit invocations of each others operations. These invocations happen on behalf of events which are announced by components and are dispatched (usually by a central managing facility) to those components which have registered for the events. These mechanisms are used, for example, in software development environments (Reiss, 1990; Bischofberger et al., 1995). Formalizations of this style have been presented by Garlan & Notkin (1991) and Luckham et al. (1995).

Systems based on the architectural style of Object-Orientation (or Data Abstraction as in (Shaw, 1995)) can greatly benefit from the use of implicit invocation mechanisms. Basically, a running object-oriented system can be viewed as graph of interconnected objects. Objects make use of other objects and delegate work to them. Delegating work to another object often means becoming dependent on this object, more precisely becoming dependent on this other object's abstract state as declared in its interface. Since an object can be referenced (aliased) by more than one object, it may change with some of the dependent objects not knowing about this change. As a consequence, these objects might get out of synchronization with the changed object and have to be updated.

If the changed object were to inform its dependent objects by explicit operation invocations, the changed object's interface and implementation would become dependent on its depending objects as well. This introduces cyclic dependencies and should be avoided since it hampers system evolution and maintenance (Sullivan & Notkin, 1992).

These dependencies can be avoided by using implicit rather than explicit invocations of the dependent objects' operations. The Event Notification pattern presented in this paper shows how this integration can be achieved smoothly.

2.1 Basic Integration Rationale

Every object has an *abstract state* declared in its interface. An object's abstract state might change due to a mutating operation call on it. This leads to a visible *abstract state change*. These changes might be described in terms of a formal object model, for example by using finite state machines or more advanced modeling techniques (Harel et al., 1990; Coleman et al., 1992; Booch, 1994).

An implicit invocation mechanism is integrated with an object-oriented system using the notion of event: an *event* represents a state change of a particular object. Each type of event represents a different type of state change within an object's abstract state model. Whenever an object is changed, it announces the event corresponding to the state change by triggering an *event notification* causing dependent objects to take notice of the state change.

An event notification is essentially the same as the implicit invocation of predefined operations of dependent objects. While an event notification is the major coordination mechanism in a system solely based on the Implicit Invocation style, in object-oriented systems it is often accompanied by an explicit invocation mechanism of the opposite direction: An object is particularly interested in receiving event notifications from those objects which it makes use of, because making use of them very often means becoming dependent on them. The dependent object has to be informed about state changes of those objects.

Therefore, we introduce implicit invocation to object-oriented systems as a means of managing dependencies between objects to ensure that they are in a consistent state. This is achieved through an event notification mechanism which takes care that dependent objects are notified about changes of those objects on which they depend. They can subsequently update themselves.

There are other ways of using event notification mechanisms, for example in distributed systems where events may be used to indicate that a component is still "alive." The major application of events in the object-oriented systems we are dealing with, however, concentrates on the problems of maintaining inter-object update dependencies, so we chose to focus on this problem.

2.2 Summary of the Event Notification Pattern

Several implementations of implicit invocation mechanisms have been presented, for example (Notkin et al., 1993; Garlan & Scott, 1993). The Event Notification pattern takes stock of these implementations and elaborates them based on our own experience. It is discussed in the next section and is based on the following rationale:

An object's abstract state is made explicit by the usual access operations that let dependent objects query its state. Possible state changes are made explicit as *state change objects* that are publicly accessible in the object's interface. Dependent objects can register to state change objects which causes them to be notified in case the event notification is triggered by the state change object's owner.

State change objects cannot be freely introduced . They have to correspond to the state changes which they represent, possibly using an underlying formal abstract state model.

A dependent object makes its dependencies explicit by *event stub objects* in its interface which represent its dependencies as first class objects. Notations for programming languages, module interconnection languages and software architecture have long made dependencies explicit as imports in module or component interfaces (Wirth, 1982; Prieto-Diaz & Neighbors, 1986; Shaw et al., 1995). Event stubs extend this concept to event notification systems.

Event stubs are linked to state change objects of other objects either directly or by using one or more intermediate *event link objects*. An event link object propagates the event notification to the dependent object. The possible chain of event links has to end with a dependent object's event stub which is a special event link itself.

This pattern lets developers decouple dependent objects from those objects which they depend on, lets them make the abstract state and dependencies of objects explicit in class interfaces, lets them dynamically and selectively register dependents for state changes and lets them encapsulate the notification process by event link objects.

2.3 Comparison with the Observer Pattern

The Event Notification pattern is similar to the Observer pattern as presented in Gamma et al. (1995). The structure of the Observer pattern is based on the original Smalltalk-80 change/update mechanism (Goldberg & Robson, 1989). Dependent objects, called observers, offer an `update:` operation which is invoked by objects on which they depend. These objects are called subjects, and they call `update:` on their observers whenever a state change occurs.

Though the Observer and Event Notification pattern have the same overall rationale, they differ in some important respects:

- The Event Notification pattern is based on an abstract state model, making possible state changes and dependencies on these state changes explicit as first class objects. This vital information is not visible in the structure of the Observer pattern, but is buried in the implementation code.
- Dependent objects in the Event Notification pattern may selectively register for certain state changes. They are only notified if this particular state change occurs. Dependent objects in the Observer pattern are always notified if a state change occurs, regardless whether they are interested or not. It should be noted that Gamma et al. suggest to enhance the basic Observer pattern with selective registration for events. This emphasizes the importance of the possibility to selectively register for certain events.
- In the Observer pattern, a single predefined operation is invoked which has to carry out the event dispatch to an appropriate operation to handle the event. In the Event Notification, the dispatch is carried out beforehand through selectively registering at state change objects via event stubs.
- The Event Notification has an event stopping behavior. Since objects selectively register for state changes, it is hard to use the event notification mechanism as a Chain of Responsibility (Gamma et al., 1995). The Observer pattern lets developers provide a class with a default forwarding behavior for events so that a Chain of Responsibility can be implemented easily.

Nevertheless, both the Event Notification and Observer pattern are similar in their overall goals. It depends on the level of abstraction chosen to look at a pattern to decide how important certain differences are. On a software design level the Observer and Event Notification pattern are different patterns since they have different design structures. On a more abstract software architecture level they serve similar purposes and thus might be treated as a single but then rather general pattern. The presentation of the Event Notification pattern in the next section focuses on the software design level, just like the Observer pattern in Gamma et al. (1995).

3 The Event Notification pattern

This section presents the Event Notification pattern. It draws on implicit invocation mechanisms (Garlan & Notkin, 1991; Notkin et al., 1993). The presentation form is based on Gamma et al. (1995).

Intent

Manage update dependencies between objects by introducing an event notification mechanism. Make the state model and dependencies thereon explicit in class interfaces to achieve transparency.

Also Known As

Implicit Invocation mechanism.

Motivation

An object which delegates work to another object by using its services becomes dependent on it. More precisely, it becomes dependent on the other object's abstract state as defined in its interface. Therefore, it must ensure its own consistency with respect to the objects on which it depends. Sometimes it is sufficient for the dependent object to poll the other object's state at certain predefined points in time. However, for a large number of systems including real-time systems, interactive applications and software development environments, this is not an adequate solution.

It is also not an adequate solution to let the changed object explicitly invoke operations of the dependent objects to inform them about a change of state, since this would intertwine the changed object with its dependent objects.

The problem shows up, for example, in interactive systems where the change of an object's state has got to lead to immediate updates in the user interface. A simple MVC based application design (Krasner & Pope, 1988) might consist of several view objects and a model object. The view objects are user interface objects that view parts of the model object which contains the application data.

Consider such a model object representing a spreadsheet. Its class interface has access operations that return the contents of a cell as well as the results of some predefined computations on the spreadsheet. A data typist enters incoming data into the model. The model object may only partially be in a consistent state raising exceptions if cells or computation results are accessed that aren't available yet.

The data typist might have the tabular view of a spreadsheet. A manager might only be interested in the result of the computations based on this spreadsheet and thus might have only a very simple interface viewing these results as simple data. A third person, an analyst, might only be interested in the differential changes within the rows and columns of the spreadsheet compared to older spreadsheets. He or she might wish to see them as graphical output on a canvas.

Figure 1 shows a rough software design. It shows the graphical notation used throughout the paper: A rectangle represents a class and an arrow represents a use-relationship. A line with a triangle as depicted in figure 4 shows an inheritance relationship.

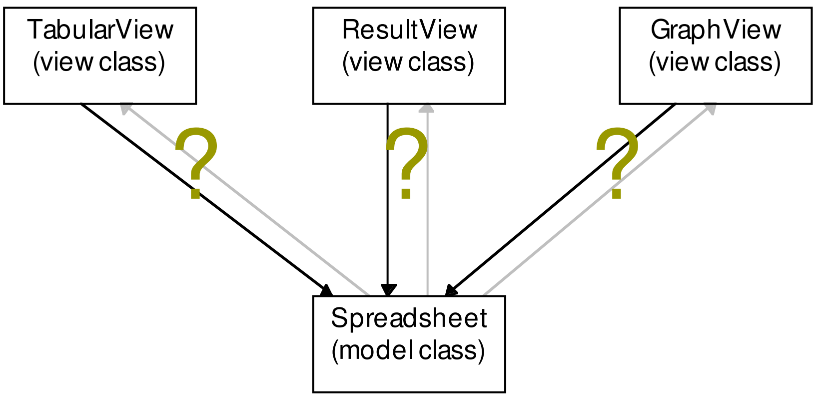

Figure 1: The software design corresponding to the three views and the single model object. The unclear dependencies and control flow have been marked with a question mark.

This simple application sufficiently demonstrates the relevant problems. There are three dependent view objects, the tabular view, the computed results and the graphical visualization, which all depend on a single model object. The dependencies differ, since the data typist might wish to see only the spreadsheet, the manager only the computed results as more become available and the analyst only the changes within the data when they occur.

The problem can be solved with the help of the *Event Notification pattern*. It is based on an abstract state model and dependencies thereon and introduces facilities to manage these dependencies using implicit invocation.

The model object explicitly declares its abstract state through the usual access operations and through *state change objects* which represent the possible state changes within the object's state model. The view objects explicitly declare their possible dependencies on the model object through *event stub objects* in their interface. Either the view objects themselves or a third party object links these event stubs to the corresponding state change objects.

The spreadsheet object, for example, offers a state change object in its interface that represents the state changes of a single spreadsheet cell. The spreadsheet might offer some more state change objects to represent the result changes computed by the spreadsheet. The tabular and graphical view objects create event stubs to link themselves to the state change object representing a cell's state change. The result view object may link itself to some state change objects representing changes in the computed results.

If a state change occurs within the model object the corresponding state change object is triggered. The state change object in turn triggers the event stubs it holds which then call a predefined operation on their owners, the views. For example, the tabular and graphical view objects are notified if a spreadsheet cell is changed. Figure 3 shows an interaction diagram for the dynamics of this process.

Using the terminology of the Observer pattern, the view objects are called the *observers* of the model object which is called the *subject*. The subject doesn't have to know about its observers but uses the event notification mechanism to notify them about changes of state.

We say that an event that has happened if an object has changed its abstract state. This is usually caused by a mutating operation call on the object. The event leads to the notification of dependent objects to inform them about the state change. This notification is essentially the implicit invocation of predefined operations of the dependent objects.

Applicability

The Event Notification pattern is used to integrate the architectural style of Implicit Invocation with the architectural style of Data Abstraction, here more precisely with Object-Orientation.

It can be used whenever the following situations occur:

- An object depends on a number of other objects and has to be notified in case of state changes of these objects as soon as possible (that is, polling is not an option).
- An object wants to be informed about specific state changes of other objects rather than every occuring state change.
- The notification process due to an object's change of state has to be carried out anonymously, that is the predefined dependent's operations have to be implicitly invoked.
- Observers and subjects have to be statically decoupled, for example, because they stem from different libraries so that the subject cannot make any assumptions about its observers.

Structure

Figure 2 shows the structure of the Event Notification pattern.

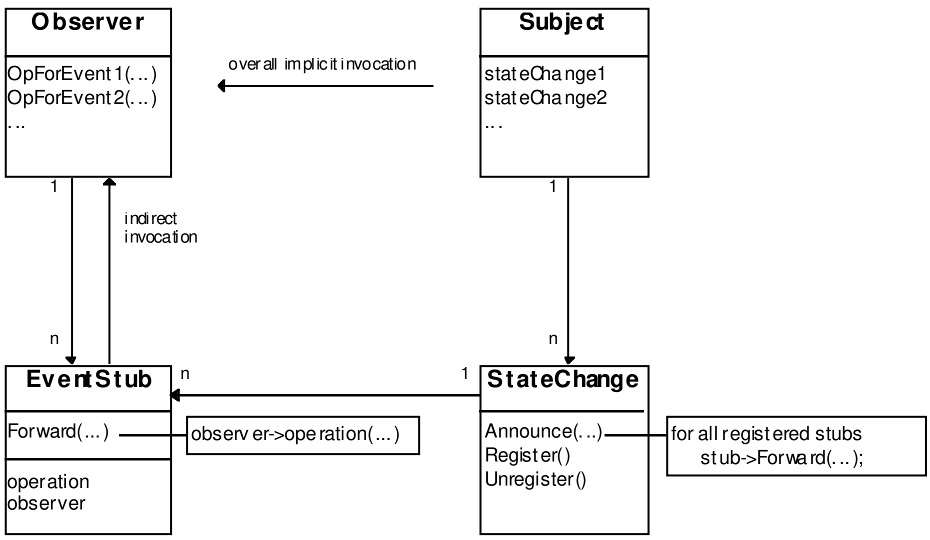

Figure 2: Class diagram of the Event Notification pattern. A Subject instance holds several state change objects in its interface which forward a notification to an Observer's event stub.

Participants

The following participants shown in the structure diagram of figure 2 can be identified:

Subject

- defines its abstract state in terms of the usual access operations and state change objects.

StateChange

- represents a possible state change within a subject's abstract state model.
- offers operations for the subject to trigger the event notification.
- offers operations to register and unregister observers via event stub objects.

Observer

- declares its dependencies on other objects abstract state by event stubs in its interface.
- provides the event stubs with an operation reference to be called in case of invocation.

EventStub

- represents a dependency within an object's abstract state model as a first class object.
- knows which operation of its owner, an observer, is to be called in case of invocation.

Collaborations

- If a subject changes its state, it triggers the corresponding state change object.
- A state change object distributes the notification to all its event stubs.
- An event stub forwards a notification to its owner, an observer.

The interaction diagram shown in figure 3 describes the dynamics of an event notification for the motivating example. Assume that the tabular view object changes a cell of the spreadsheet using explicit invocation. The spreadsheet then triggers the state change object representing the state change of cells. The state change object forwards the notification to the event stubs of the tabular and graphical view objects which have registered for this particular event. The event stubs forward the notification to a predefined operation of their respective owners.

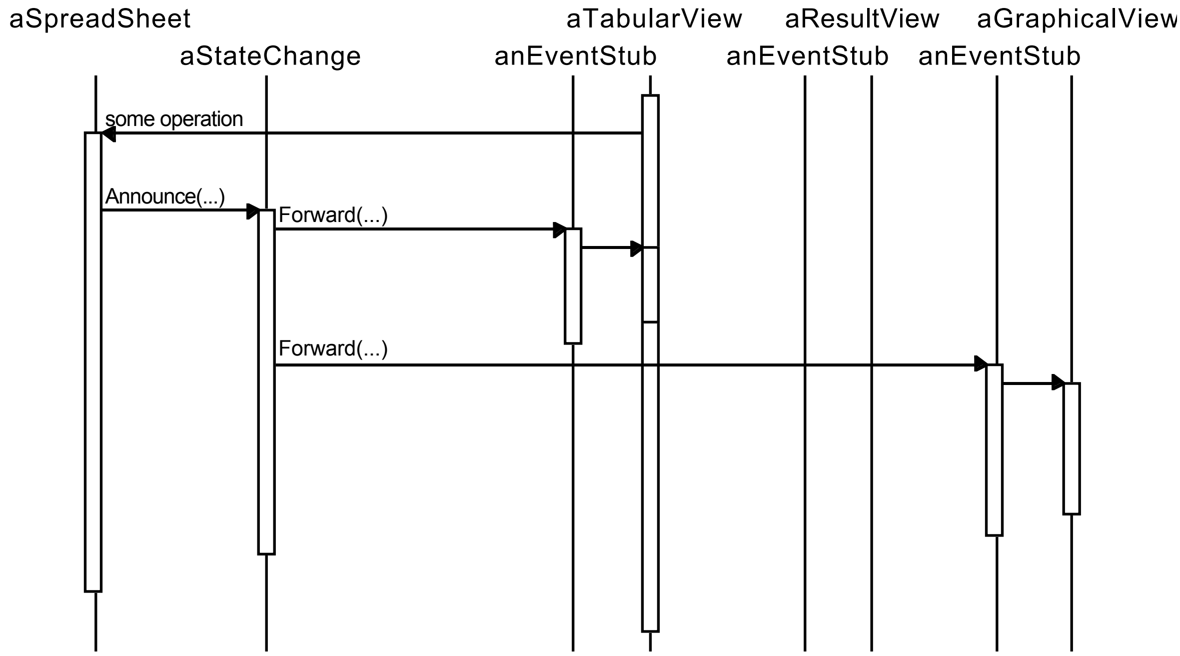

Figure 3: Interaction diagram for the motivating example. It illustrates the case of a changed spreadsheet cell.

Consequences

The Event Notification pattern lets developers decouple observers and subjects while preserving the vital dependency relationship needed to provide immediate feedback and to maintain system consistency. In particular, the Event Notification pattern has the following advantages, responsibilities and consequences.

1. *Static decoupling of observer and subject.* The subject need not assume anything about its observers. In particular, the event notification does not depend on a specific superclass the observer has to inherit.
2. *Explicit modeling of abstract state.* Explicit modeling of a subject's abstract state both by access operations and state change objects helps to clarify class semantics. Possible events are made explicit as state change objects and are not buried in the implementation.
3. *Explicit modeling of dependencies.* The dependencies on other objects are modeled as event stubs in an object's class interface and are therefore explicit as well.
4. *Derivation of state change objects.* The number of different state change objects is fixed and can be derived from the abstract state model, at least those parts that can be modeled formally.
5. *Selective registration.* An observer selectively registers for certain state changes. Thus, it does not have to test for events it is not interested in. An observer decides in advance whether it will be notified about an event or not.
6. *Early dispatch.* Dispatching of the event associated with a state change is implicitly done by the event stub object which calls its predefined operation.
7. *Event stopping behavior.* Since the observer receives notifications only about those events which it has explicitly registered for, it is difficult to implement a Chain of Responsibility using the implicit invocation mechanism. This has to be implemented as a separate protocol.
8. *Third party configuration.* Since observers are fully decoupled and rely in principle only on event stub objects, the dependency relationship can be configured easily by third party objects.

Implementation

The following issues should be considered when implementing the Event Notification pattern. Some of these issues can also be found in the discussion of the Observer pattern in Gamma et al. (1995).

1. *Reusable StateChange and EventStub classes.* The key to a straightforward implementation of the Event Notification pattern is to design reusable state change and event stub classes. In dynamically typed languages like Smalltalk or CLOS this doesn't pose much problems. It can also be done in statically typed programming languages like C++ without much overhead, as long as parameterized types and class-bound operation references are available (see the sample code).
2. *Abstracting the notification protocol.* It is best to introduce a general EventLink class of which EventStub is only a particular subclass. The general EventLink class defines the protocol for forwarding an event notification. Subclasses implement the way this forwarding is carried out in different ways, serving different purposes. A revised structure diagram for this enhancement is shown in figure 4. A class IACEventLink, for example, might implement the forwarding process by using a remote procedure call or any other interapplication communication facility. IACEventLink serves to make the notification transparent to process boundaries.

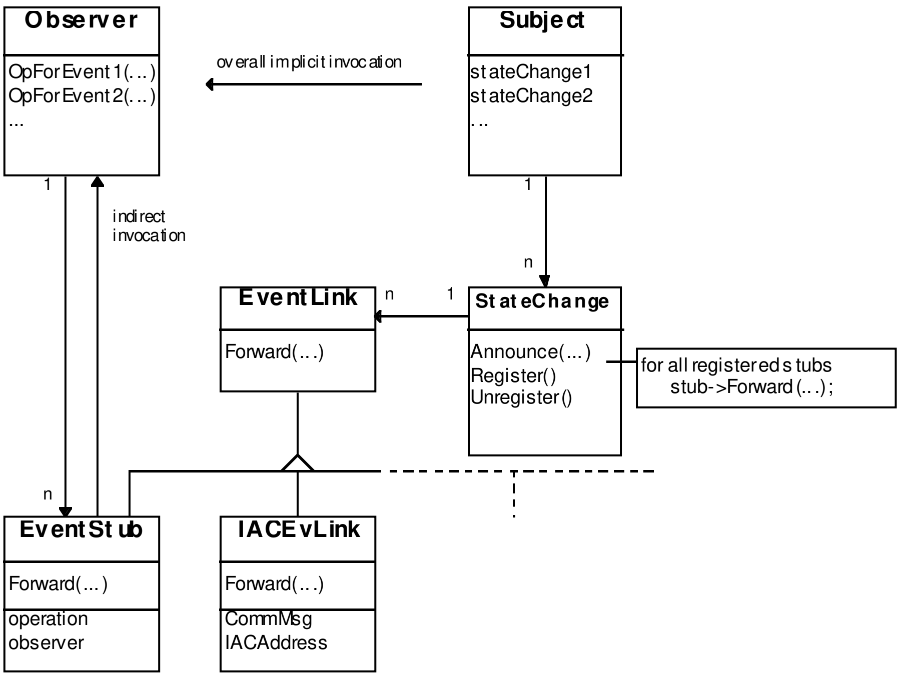

Figure 4: Revised structure diagram, showing the abstract class EventLink and its specializations EventStub and IACEventLink (inheritance is indicated through the triangle on the line connecting the classes).

3. *Chain of event links.* At runtime, several event links might be concatenated, so that a chain of event links results. It must end with an event stub that finally propagates the notification to an observer. The purpose of an event link chain is to forward a notification to a single observer (which nevertheless might be a proxy in a different process that dispatches the event notification to a set of local observers).

4. *Guidelines for Observer/Subject design.* If not implemented carefully, observers and subjects might be captured in an infinite change/update loop. An observer changing a subject might receive an event notification leading to further changes of the subject leading to event notifications and so forth.

   Therefore: always design and implement an observer in such a way that it reacts to a notification only with non-mutating operations on the subject. As a consequence, each subject has to clearly separate mutating from non-mutating operations in its interface. This rule is transitive which means that observers shouldn't change those objects which are predecessors of their subjects in a notification process possibly consisting of several stages.

5. *Parameterization of event notification.* The parameterization of the event notification can be chosen according to a system's need. This is a difficult task, however, since the subject has to define the protocol of the notification process without making undue assumptions about its context of use. Among the possible parameterizations two stand out. Both variants can coexist, possibly as two state change objects representing the same state change:

   - No parameters (or just the subject and an event identification) are passed. With no state change specific parameters the observer has to retrieve further state information from the subject, which is possibly problematic in a network context and results in performance penalties. This parameterization makes sense, if an observer seldom retrieves further data.
   - All data affected by the state change are passed. This fully informs the observers about the outcome of a state change so that they usually don't have to request further information from the subject.

6. *Concurrency.* Design issues like concurrency are not covered by this pattern but have to be made explicit in any actual design. However, the order of notifications should not introduce hidden dependencies (see guidelines for Observer and Subject design). The observers should be independent from each other with respect to notifications received from a subject.

7. *Dependency Manager.* The relationships between event stub and state change objects can be maintained by a single managing facility so that not every state change object has got to have a collection of event links of its own. By doing so memory consumption can be reduced.

8. *Event history.* It might make sense to log the events announced by an object so that observers that startup late can catch up with the object. Otherwise they will have to query its state, which may not be appropriate for objects encapsulating large amounts of data.

Sample Code

In the following, two examples are presented. The first one illustrates the use of a parameterless `EventStub` and `StateChange` object for decoupling a simple `Counter` object from its `CounterView`. This example demonstrates the basic relationships that structure implementations of the pattern. The second example demonstrates the power of using parameterized types to implement the pattern by describing an actual design for the motivation section.

The following lines of code show the classes `Counter` and `CounterView`. A counter maintains and possibly increments a counter. The `CounterView` serves to display the counter value on a screen.

```
class Counter
{
public:
   StateChange* scChanged;
   virtual int Value();
   virtual void Increase();
   Counter();

private:
   int _value;
};

class CounterView
{
public:
   CounterView(Counter*);
   virtual void Update();

private:
   Counter* _counter;
   EventLink* _link;
};
```

The `CounterView` makes direct use of the `Counter` interface. The counter, however, doesn't know about particular views but informs them via the `StateChange` object defined in its interface. The counter view links itself to the counter using an `EventStub` object. The interfaces and implementations of these classes are defined as follows:

```
class EventLink
{
public:
   virtual void Forward() = 0;
};


template<class T>
class EventStub : public EventLink
{
public:
   EventStub(T* obs, void(T::*fun)()) {
      _obs = obs; _fun = fun;
   }
   virtual void Forward() {
      (_obs->*_fun)();
   }

private:
   T* _obs;
   void (T::*_fun)();
};


class StateChange
{
public:
   virtual void Register(EventLink* link) {
      evLinks->Append(link);
   }
   virtual void Unregister(EventLink* link) {
      evLinks->Remove(link);
   }
   virtual void Announce() {
      Iter<EventLink*> iter(evLinks);
      while (EventLink* evLink = iter()) {
         evLink->Forward();
      }
   }
   StateChange::StateChange() {
      evLinks = new List<EventLink*>();
   }

private:
   List<EventLink*>* evLinks;
};
```

The class EventLink defines a general observer independent protocol to receive an event notification. Its subclass EventStub is a parameterized class which takes as an argument the type of observer. This turns it into a concrete class specialized for a specific observer class. A concrete EventStub object is hooked on a StateChange instance using the Register operation. The StateChange object keeps track of EventLink objects so that it doesn't know about observer classes.

The following code links the Update operation of CounterView to the state change object scChanged of Counter:

```
CounterView::CounterView(Counter* counter)
{
   _link = new EventStub<CounterView>(
      this, &CounterView::Update);
   _counter = counter;
   counter->scChanged->Register(_link);
}
```

The creation process for an EventStub instance consists of two steps. First, the template is instantiated with the concrete Observer class as a formal parameter, leading to a concrete class from which an object can be created. This is carried out using new, and the new object receives a reference to the observer and its operation to be called in case of an event notification. Finally, the CounterView hooks this EventStub on the scChanged object thereby establishing the link to the Counter object.

The decoupling achieved here is unidirectional: CounterView knows about Counter, but not vice versa. Had the CounterView made the EventStub object available in its public interface, a third party object could have taken it and linked it to any Counter without the CounterView knowing about it.

Now the more complex example from the motivation section is considered. The following code shows the interface of a spreadsheet class which is to be observed by different kinds of views:

```
class Spreadsheet
{
public:
      // operations for changed cell
   virtual bool IsValidCell(int x, int y);
   virtual Item* GetCell(int x, int y);
   StateChange3<Item*, int, int>* scCellChanged;
   virtual void Enter(Item*, int x, int y);

      // computable result 1
   virtual bool IsResult1Valid();
   virtual Item* GetResult1();
   StateChange1<Item*>* scResult1Valid;

      // computable result 2
   virtual bool IsResult2Valid();
   virtual Item* GetResult2();
   StateChange1<Item*>* scResult2Valid;

      // further operations ...
   virtual void Run();
   Spreadsheet();

private:
   Item _cells[64][64];
   Item _result1;
   Item _result2;
};
```

Spreadsheet offers three state change objects which observers can hook their event stubs on:

- scCellChanged of type StateChange3<Item*,int,int> to notify observers about changes to a cell in the spreadsheet,
- scResult1Valid of type StateChange1<Item*> to notify observers when result1 becomes available, and
- scResult2Valid, also of type StateChange1<Item*>, to notify observers when result2 becomes available.

These classes were instantiated from the template classes StateChange1 and StateChange3 which are defined as follows:

```
template<class Arg1>
class StateChange1
{
public:
   virtual void Register(EventLink1<Arg1>*);
   virtual void Unregister(EventLink1<Arg1>*);
   virtual void Announce(Arg1 a1);
   StateChange1();

private:
```

```
   List<EventLink1<Arg1>*>* evLinks;
};


template<class Arg1, class Arg2, class Arg3>
class StateChange3
{
public:
   virtual void Register(
      EventLink3<Arg1, Arg2, Arg3>* link);
   virtual void Unregister(
      EventLink3<Arg1, Arg2, Arg3>* link);
   virtual void Announce(
      Arg1 a1, Arg2 a2, Arg3 a3);
   StateChange3();

private:
   List<EventLink3<Arg1, Arg2, Arg3>*>* evLinks;
};
```

The parameters Arg1 to ArgN for a class StateChangeN are formal parameters of the template to be substituted with the types of objects passed along with a concrete event notification. These types are derived from the state transitions in the abstract state space of the spreadsheet class which the `StateChange` objects represent. So, if the status of `result1` changes from invalid to computed (and thus valid), the corresponding flag switches to true and the variable holding `result1` is set to its computed value. For pragmatic reasons it is assumed that it cannot become invalid any more so that it suffices to parameterize the event notification with the new result value only (omitting the flag).

For each `StateChangeN` there is a corresponding `EventStubN` class which is a subclass of `EventLinkN` similar to the `Counter` example above:

```
template<class Arg1>
class EventLink1
{
public:
   virtual void Forward(Arg1) = 0;
};


template<class T, class Arg1>
class EventStub1 : public EventLink1<Arg1>
{
public:
   EventStub1(T* obs, void(T::*fun)(Arg1));
   virtual void Forward(Arg1 a1);

private:
   T* _obs;
   void (T::*_fun)(Arg1);
};


// corresponding classes for
// EventLink2/EventStub2,
// EventLink3/EventStub3,
// etc.
```

Observers like `GraphView`, `TabularView` or `ResultView` use these event link classes to hook themselves onto the state change objects of `Spreadsheet`. Assuming that the class `TabularView` is interested in all three state changes declared by `Spreadsheet`, it has to create three corresponding event stub objects, called `esCellChanged`, `esResult1Valid` and `esResult2Valid` below. The initialization code for `TabularView` might look like the following:

```
TabularView::TabularView(Spreadsheet* ss)
{
   esCellChanged =
      new EventStub3<TabularView, Item*, int, int>(
         this, &TabularView::CellChanged);
   ss->scCellChanged->Register(esCellChanged);

   esResult1Valid =
      new EventStub1<TabularView, Item*>(
         this, &TabularView::Result1Valid);
   ss->scResult1Valid->Register(esResult1Valid);

   esResult2Valid =
      new EventStub1<TabularView, Item*>(
         this, &TabularView::Result2Valid);
   ss->scResult2Valid->Register(esResult2Valid);
}
```

Here, the TabularView registers its event stubs with all three state change objects of Spreadsheet. It is also possible to fully decouple an observer from a subject. The following code hooks the `GraphView` onto the `Spreadsheet` without them knowing each other.

```
EventLink3<Item*, int, int> evStub =
   graphView->esValueChanged;
StateChange3<Item*, int, int> sChange =
   spreadsheet->scCellChanged;

sChange->Register(evStub);
```

To the `GraphView` it looks like an unknown source provides it with values from a two-dimensional matrix. Thus, the spreadsheet can be replaced easily with any other source that follows this event notification protocol.

Announcing that a cell has changed is straightforward. The spreadsheet class simply calls:

```
scCellChanged.Announce(&_cells[x][y], x, y);
```

`Announce` forwards this call and the notification parameters to all registered event links which, being event stubs in this case, forward the notification directly to the observers they were created by.

Known Uses

Different implementations of this pattern have been reported in (Garlan & Scott, 1993; Notkin et al., 1993). The pattern as presented here has been used in our own framework (Riehle & Züllighoven, 1995).

Related Patterns

This pattern is closely related to the Observer pattern and Implicit Invocation (Gamma et al., 1995; Notkin et al., 1993). It is often used in conjunction with the Mediator pattern (Gamma et al. 1995; Sullivan & Notkin 1992; Sullivan 1994) which integrates and mediates between otherwise independent objects.

4 Related Work

The discussed implementation of the Event Notification pattern is related to the Implicit Invocation implementations presented in (Garlan & Scott, 1993; Notkin et al., 1993; Sullivan & Notkin, 1992). Classes of those objects which have been enhanced with an event notification mechanism have been called *Abstract Behavior Types* by Sullivan (1994).

The model presented here elaborates the object-oriented runtime model further. It makes dependencies on events explicit in class interfaces and represents the connector linking the objects as a first class object as well.

The referenced implicit invocation mechanisms for statically typed languages rely on additional tools (code generators or macro preprocessors) to introduce events to software modules. Using parameterized types as presented avoids the need for additional tools and greatly eased our programming efforts. It made our programs type-safe while solving typing issues considered to be unclear in Notkin et al. (1993).

The OMG has specified an event service model for CORBA (OMG, 1993), which is similar to the pattern presented here. The CORBA event service specification is more general in that it leaves open several design choices that have been decided by the pattern in advance. The CORBA model accepts both typed and untyped events while the preferred pattern implementation explicitly introduces static typing to event notifications.

The CORBA model uses event channels to process and forward events to clients. The forwarding can be synchronous or asynchronous. An event channel can distribute an event to more than one receiver.

The Event Notification pattern uses event links to forward events to dependent objects. Event links can be used for both synchronous and asynchronous communication. The pattern, however, doesn't use event links to distribute an event to its receivers but expects different receivers to register directly at a state change object.

5 Conclusions

We have demonstrated that Implicit Invocation and Object-Orientation can be integrated well. We have shown how to do this by discussing the Event Notification pattern, a pattern with a clear rationale based on an object's abstract state model. This rationale makes possible state changes of an object, dependencies thereon and the runtime links between an object and its dependents explicit as first class objects.

We have further shown that the pattern can be implemented efficiently and in a type-safe fashion by using parameterized types. It thereby overcomes some problems of earlier implementations of implicit invocation mechanisms. A more elaborate version of the sample code from section 3 has been compiled and tested and can be obtained from the author.

It is interesting to note that two patterns with a similar overall rationale (Observer and Event Notification) have evolved almost independently of each other and lead to such different structures. It might be worthwhile therefore to understand and explore patterns as a (software) cultural phenomenon, with concrete patterns arising always on a specific background of experience. Given this, the task of spanning different communities becomes relevant and the resulting integration work might be one of the interesting challenges the patterns community should undertake in the future.

Acknowledgments

I wish to thank Steve Berczuk, Kai-Uwe Mätzel, Stefan Roock, Douglas Schmidt, Wolfgang Strunk and Henning Wolf for reviewing and commenting on the paper thereby helping me to improve it.

I would further like to thank Karl-Heinz Sylla for reviewing and commenting. He originally pointed out to me the guidelines for Observer and Subject design in section 3.

Bibliography

Bischofberger, W., Kofler, T., Mätzel, K.-U. & Schäffer, B. (1995). Computer Supported Cooperative Software Engineering with Beyond-Sniff. SEE '95, Conference Proceedings, 135-143.

Booch, G. (1994). Object-Oriented Analysis and Design with Applications. 2nd Edition. Redwood City, California: Benjamin/Cummings.

Coleman, D., Hayes, F. & Bear, S. (1992). Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design. IEEE Transactions on Software Engineering 18 (1), 9-18.

Coplien, J., & Schmidt, D. (Eds.). (1995). Pattern Languages of Program Design. Reading, Massachusetts: Addison-Wesley.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). Design Patterns: Elements of Reusable Design. Reading, Massachusetts: Addison-Wesley.

Garlan, D. & Notkin, D. (1991). Formalizing Design Spaces: Implicit Invocation Mechanisms. LNCS 551, VDM '91, Conference Proceedings, 31-44.

Garlan, D. & Scott, C. (1993). Adding Implicit Invocation to Traditional Programming Languages. ICSE-15, Conference Proceedings, 447-455.

Garlan, D. & Shaw, M. (1993). An Introduction to Software Architecture. In Advances in Software and Knowledge Engineering, Volume 2. World Scientific Publishing Company. See also: CMU Technical Report CMU-CS-94-166. Pittsburgh, Pennsylvania: Carnegie Mellon University.

Goldberg, A. & Robson, D. (1989). Smalltalk-80: The Language. Reading, Massachusetts: Addison-Wesley.

Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Trauring, S., & Trakhtenbrot, M. (1990). STATEMATE: A Working Environment for the Development of Complex Reactive Systems. IEEE Transactions on Software Engineering 16 (4).

Krasner, G. E. & Pope, S. T. (1988). A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. Journal of Object-Oriented Programming 1 (3), 26-49.

Luckham, D. C., Kenney, J. J., Augustin, L. M., Vera, J., Bryan, D. & Mann, W. (1995). Specification and Analysis of System Architecture Using Rapide. IEEE Transactions on Software Engineering 21 (4), 336-355.

Notkin, D., Garlan, D., Griswold, W. G. & Sullivan, K. (1993). Adding Implicit Invocation to Languages: Three Approaches. LNCS 742, ISOTAS '93, Conference Proceedings, 489-510.

OMG. (1993). Event Service Specification. OMG TC Document 93.7.3.

Prieto-Diaz, R. & Neighbors, J. M. (1986). Module Interconnection Languages. Journal of Systems and Software 6 (4), 307-334.

Reiss, S. P. (1990). Connecting Tools Using Message Passing in the Field Environment. IEEE Software, 57-66.

Riehle, D. & Züllighoven, H. (1995). A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor. In Coplien & Schmidt (1995), 9-42.

Shaw, M. (1995). Patterns for Software Architecture. In Coplien & Schmidt (1995), 453-462.

Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., Zelesnik, G. (1995). Abstractions for Software Architecture and Tools to Support Them. IEEE Transactions on Software Engineering 21 (4), 314-335.

Sullivan, K. (1994). Mediators: Easing the Design and Evolution of Integrated Systems. Ph.D. Thesis, Technical Report 94-08-01. Department of Computer Science and Engineering, University of Washington.

Sullivan, K. J. & Notkin, D. (1992). Reconciling Environment Integration and Software Evolution. ACM Transactions on Software Engineering and Methodology 1 (3), 229-268.

Wirth, N. (1982). Programming in Modula-2. Berlin, Heidelberg: Springer-Verlag.