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.

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.

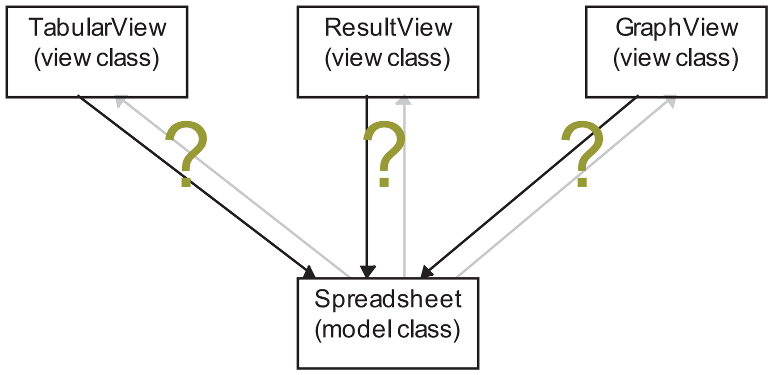


Figure 1: The software design corresponding to the three views and the single model object. The unclear dependencies and control flow have been marked with a question mark.

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

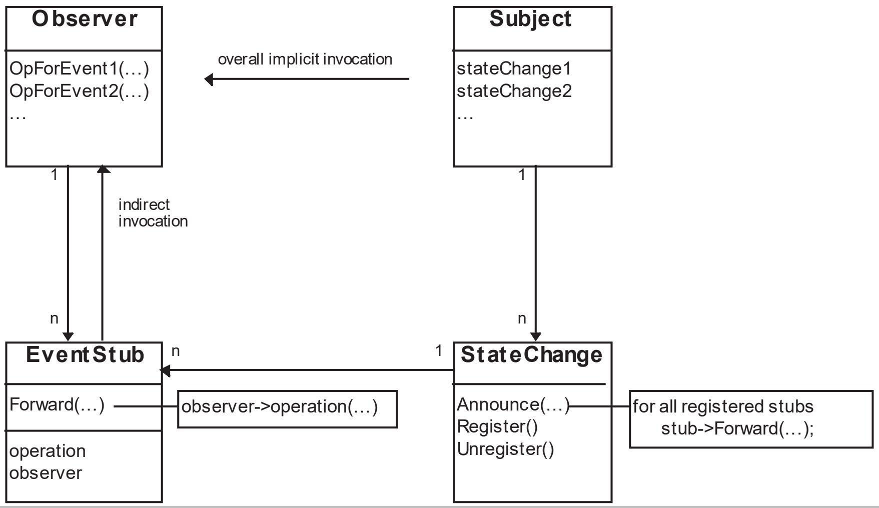
The Event Notification pattern is used to integrate the architectural style of Implicit Invocation with the architectural style of Data Abstraction, here more precisely with Object-Orientation.

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.



DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | **sales@docketalarm.com** | **1-866-77-FASTCASE**

fastcase® Smarter legal research.