Observer

Intent

Define a one-to-many dependency between objects so that when oneobject changes state, all its dependents are notified and updatedautomatically.

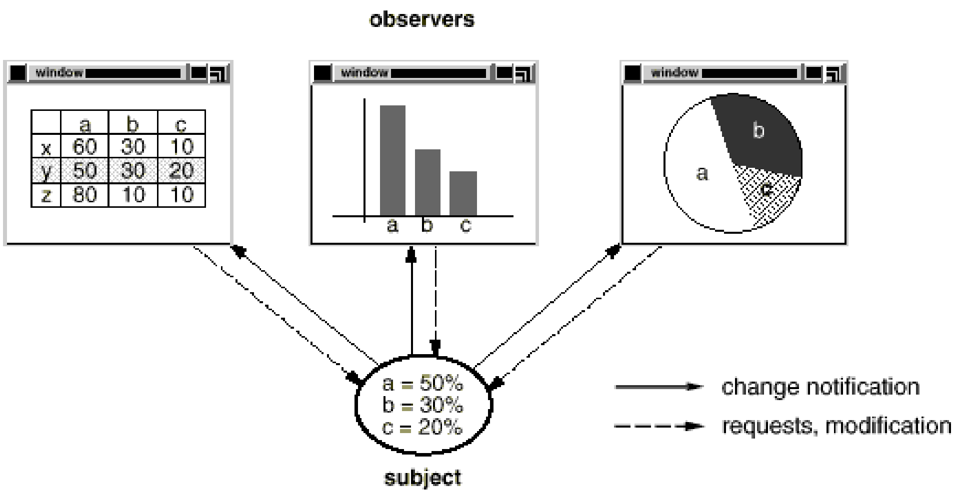
Also Known As

Dependents, Publish-Subscribe

Motivation

A common side-effect of partitioning a system into a collection ofcooperating classes is the need to maintain consistency betweenrelated objects. You don't want to achieve consistency by making theclasses tightly coupled, because that reduces their reusability.

For example, many graphical user interface toolkits separate thepresentational aspects of the user interface from the underlyingapplication data [KP88, LVC89, P+88, WGM88].Classes defining application data and presentations can be reusedindependently. They can work together, too. Both a spreadsheet objectand bar chart object can depict information in the same application dataobject using different presentations. The spreadsheet and the bar chartdon't know about each other, thereby letting you reuse only the one youneed. But they *behave* as though they do. When the user changes theinformation in the spreadsheet, the bar chart reflects the changesimmediately, and vice versa.



This behavior implies that the spreadsheet and bar chart are dependenton the data object and therefore should be notified of any change inits state. And there's no reason to limit the number of dependentobjects to two; there may be any number of different user interfacesto the same data.

The Observer pattern describes how to establish these relationships.The key objects in this pattern are **subject** and**observer**. A subject may have any number of dependentobservers. All observers are notified whenever the subject undergoesa change in state. In response, each observer will query the subjectto synchronize its state with the subject's state.

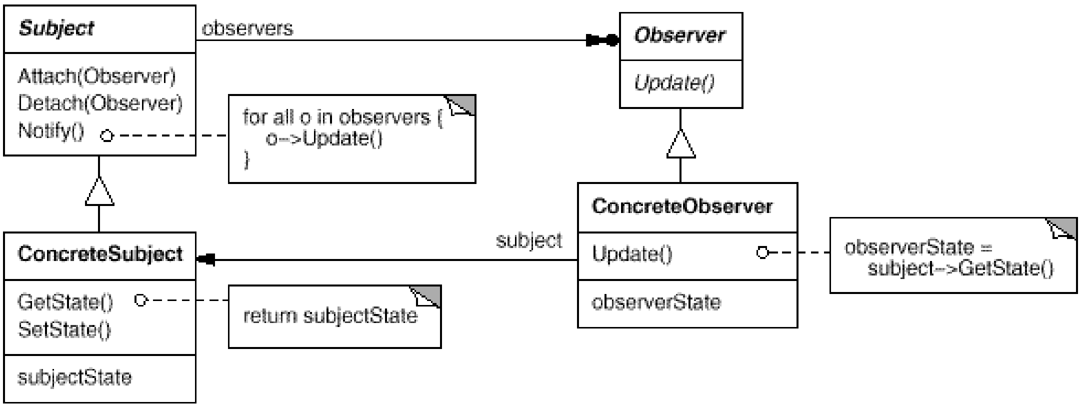
This kind of interaction is also known as**publish-subscribe**. The subject is the publisher ofnotifications. It sends out these notifications without having to knowwho its observers are. Any number of observers can subscribe toreceive notifications.

Applicability

Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other.Encapsulating these aspects in separate objects lets you vary andreuse them independently.
- When a change to one object requires changing others, and youdon't know how many objects need to be changed.
- When an object should be able to notify other objects without makingassumptions about who these objects are. In other words, you don'twant these objects tightly coupled.

Structure



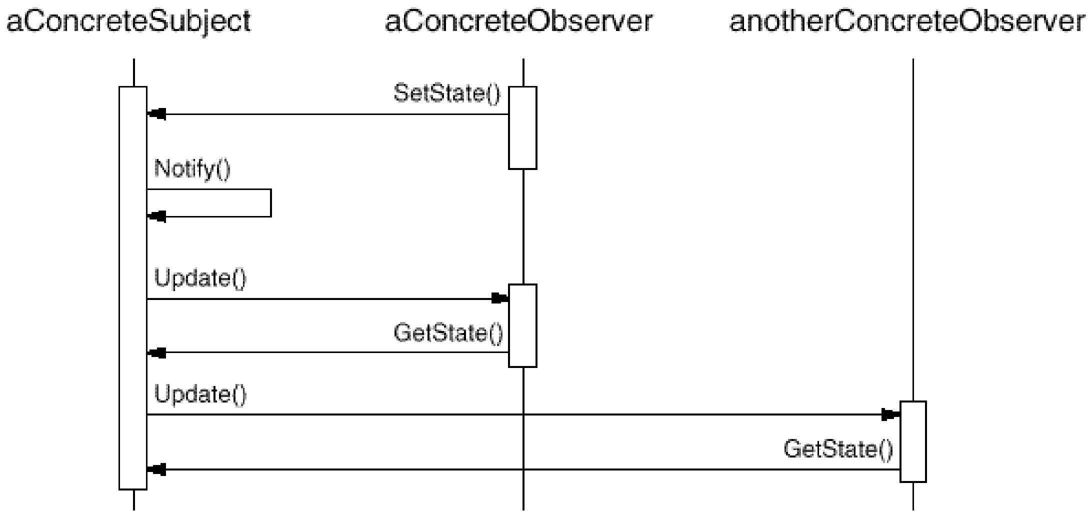
Participants

- **Subject**
  - o knows its observers. Any number of Observer objects may observe a subject.
  - o provides an interface for attaching and detaching Observer objects.
- **Observer**
  - o defines an updating interface for objects that should be notified of changes in a subject.
- **ConcreteSubject**
  - o stores state of interest to ConcreteObserver objects.
  - o sends a notification to its observers when its state changes.
- **ConcreteObserver**
  - o maintains a reference to a ConcreteSubject object.
  - o stores state that should stay consistent with the subject's.
  - o implements the Observer updating interface to keep its state consistent with the subject's.

Collaborations

- ConcreteSubject notifies its observers whenever a changeoccurs that could make its observers' state inconsistent with its own.
- After being informed of a change in the concrete subject, aConcreteObserver object may query the subject for information.ConcreteObserver uses this information to reconcile its state with thatof the subject.

The following interaction diagram illustrates the collaborationsbetween a subject and two observers:



Note how the Observer object that initiates the change requestpostpones its update until it gets a notification from the subject.Notify is not always called by the subject. It can be called by anobserver or by another kind of object entirely. The Implementationsection discusses some common variations.

Consequences

The Observer pattern lets you vary subjects and observersindependently. You can reuse subjects without reusing theirobservers, and vice versa. It lets you add observers withoutmodifying the subject or other observers.

Further benefits and liabilities of the Observer pattern include thefollowing:

1. *Abstract coupling between Subject and Observer*.All a subject knows is that it has a list of observers, eachconforming to the simple interface of the abstract Observer class.The subject doesn't know the concrete class of any observer. Thus thecoupling between subjects and observers is abstract and minimal.

   Because Subject and Observer aren't tightly coupled, they can belong todifferent layers of abstraction in a system. A lower-level subjectcan communicate and inform a higher-level observer, thereby keeping thesystem's layering intact. If Subject and Observer are lumpedtogether, then the resulting object must either span two layers (andviolate the layering), or it must be forced to live in one layer orthe other (which might compromise the layering abstraction).

2. *Support for broadcast communication.*Unlike an ordinary request, the notification that a subject sendsneedn't specify its receiver. The notification is broadcastautomatically to all interested objects that subscribed to it. Thesubject doesn't care how many interested objects exist; its onlyresponsibility is to notify its observers. This gives you the freedomto add and remove observers at any time. It's up to the observer tohandle or ignore a notification.
3. *Unexpected updates.*Because observers have no knowledge of each other's presence, they canbe blind to the ultimate cost of changing the subject. A seeminglyinnocuous operation on the subject may cause a cascade of updates toobservers and their dependent objects. Moreover, dependency criteriathat aren't well-defined or maintained usually lead to spuriousupdates, which can be hard to track down.

   This problem is aggravated by the fact that the simple update protocolprovides no details on *what* changed in the subject. Withoutadditional protocol to help observers discover what changed, they maybe forced to work hard to deduce the changes.

Implementation

Several issues related to the implementation of the dependencymechanism are discussed in this section.

1. *Mapping subjects to their observers.*The simplest way for a subject to keep track of the observers itshould notify is to store references to them explicitly in thesubject. However, such storage may be too expensive when there aremany subjects and few observers. One solution is to trade space fortime by using an associative look-up (e.g., a hash table) to maintainthe subject-to-observer mapping. Thus a subject with no observersdoes not incur storage overhead. On the other hand, this approachincreases the cost of accessing the observers.
2. *Observing more than one subject.*It might make sense in some situations for an observer to depend onmore than one subject. For example, a spreadsheet may depend on morethan one data source. It's necessary to extend the Update interfacein such cases to let the observer know *which* subject is sendingthe notification. The subject can simply pass itself as a parameterin the Update operation, thereby letting the observer know whichsubject to examine.

3. *Who triggers the update?*The subject and its observers rely on the notification mechanism tostay consistent. But what object actually calls Notify to trigger theupdate? Here are two options:
    a. Have state-setting operations on Subject call Notify after theychange the subject's state. The advantage of this approach is thatclients don't have to remember to call Notify on the subject. Thedisadvantage is that several consecutive operations will causeseveral consecutive updates, which may be inefficient.
    b. Make clients responsible for calling Notify at the right time.The advantage here is that the client can wait to trigger the updateuntil after a series of state changes has been made, therebyavoiding needless intermediate updates. The disadvantage is thatclients have an added responsibility to trigger the update. Thatmakes errors more likely, since clients might forget to call Notify.
4. *Dangling references to deleted subjects.*Deleting a subject should not produce dangling references in itsobservers. One way to avoid dangling references is to make thesubject notify its observers as it is deleted so that they can resettheir reference to it. In general, simply deleting the observers is not an option, because other objects may reference them, or they may beobserving other subjects as well.
5. *Making sure Subject state is self-consistent beforenotification.*It's important to make sure Subject state is self-consistent beforecalling Notify, because observers query the subject for its currentstate in the course of updating their own state.

    This self-consistency rule is easy to violate unintentionally whenSubject subclass operations call inherited operations. For example,the notification in the following code sequence is trigged when thesubject is in an inconsistent state:

```
void MySubject::Operation (int newValue) {
BaseClassSubject::Operation(newValue);
// trigger notification
_myInstVar += newValue;
// update subclass state (too late!)
}
```

You can avoid this pitfall by sending notifications from template methods (Template Method (360)) in abstract Subjectclasses. Define a primitive operation for subclasses to override, and makeNotify the last operation in the template method, which will ensure thatthe object is self-consistent when subclasses override Subjectoperations.

```
void Text::Cut (TextRange r) {
```

```
ReplaceRange(r);      // redefined in subclasses
Notify();
}
```

By the way, it's always a good idea to document which Subject operationstrigger notifications.

6. *Avoiding observer-specific update protocols: the pushand pull models.* Implementations of the Observer pattern often havethe subject broadcast additional information about the change. Thesubject passes this information as an argument to Update. The amountof information may vary widely.

   At one extreme, which we call the **push model**, the subjectsends observers detailed information about the change, whether theywant it or not. At the other extreme is the **pull model**;the subject sends nothing but the most minimal notification, andobservers ask for details explicitly thereafter.

   The pull model emphasizes the subject's ignorance of its observers,whereas the push model assumes subjects know something about theirobservers' needs. The push model might make observers less reusable,because Subject classes make assumptions about Observer classes thatmight not always be true. On the other hand, the pull model may beinefficient, because Observer classes must ascertain what changedwithout help from the Subject.

7. *Specifying modifications of interest explicitly.*You can improve update efficiency by extending the subject'sregistration interface to allow registering observers only forspecific events of interest. When such an event occurs, the subjectinforms only those observers that have registered interest in thatevent. One way to support this uses the notion of**aspects** for Subject objects. To registerinterest in particular events, observers are attached to theirsubjects using

```
void Subject::Attach(Observer*, Aspect& interest);
```

   where interest specifies the eventof interest. At notification time, the subject supplies the changedaspect to its observers as a parameter to the Update operation. Forexample:

```
void Observer::Update(Subject*, Aspect& interest);
```

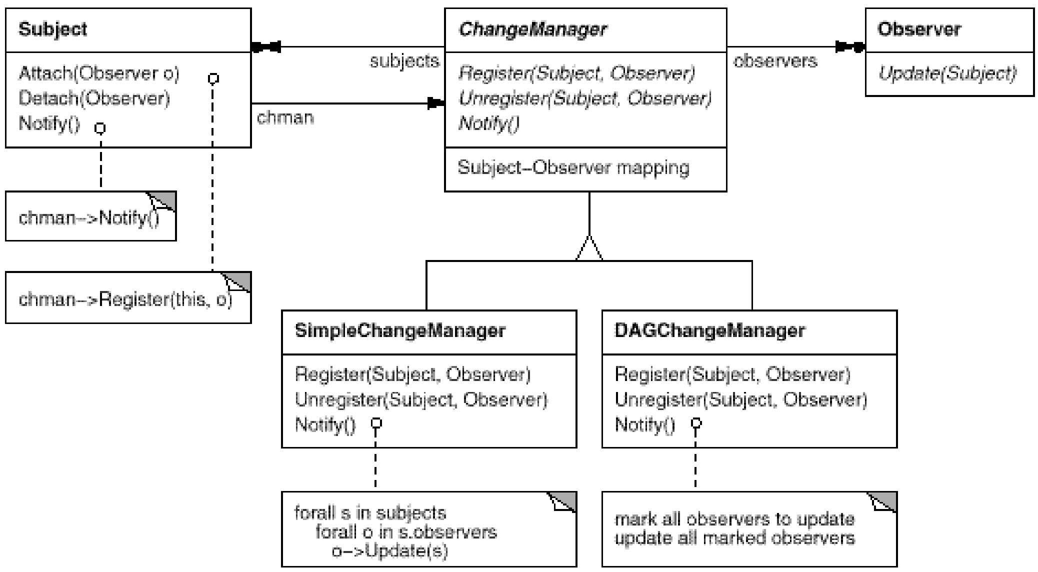
8. *Encapsulating complex update semantics.*When the dependency relationship between subjects and observers isparticularly complex, an object that maintains these relationships mightbe required. We call such an object a **ChangeManager**. Itspurpose is to minimize the work required to make observers

reflect achange in their subject. For example, if an operation involveschanges to several interdependent subjects, you might have toensure that their observers are notified only after *all* thesubjects have been modified to avoid notifying observers more thanonce.

ChangeManager has three responsibilities:

1. It maps a subject to its observers and provides an interface tomaintain this mapping. This eliminates the need for subjects to maintainreferences to their observers and vice versa.
2. It defines a particular update strategy.
3. It updates all dependent observers at the request of a subject.

The following diagram depicts a simple ChangeManager-based implementation ofthe Observer pattern. There are two specialized ChangeManagers.SimpleChangeManager is naive in that it always updates all observers ofeach subject. In contrast, DAGChangeManager handles directed-acyclicgraphs of dependencies between subjects and their observers. ADAGChangeManager is preferable to a SimpleChangeManager when an observerobserves more than one subject. In that case, a change in two or moresubjects might cause redundant updates. The DAGChangeManager ensuresthe observer receives just one update. SimpleChangeManager isfine when multiple updates aren't an issue.



ChangeManager is an instance of the Mediator (305) pattern. In general there is only one ChangeManager, and it is knownglobally. The Singleton (144) pattern would beuseful here.

9. *Combining the Subject and Observer classes.*Class libraries written in languages that lack multiple inheritance(like Smalltalk) generally don't define separate Subject and Observerclasses but combine their interfaces in one class. That lets youdefine an object that acts as both a subject and an observer withoutmultiple inheritance. In Smalltalk, for example, the Subject andObserver interfaces are defined in the root class Object, making themavailable to all classes.

Sample Code

An abstract class defines the Observer interface:

```
class Subject;

class Observer {
public:
        virtual ~ Observer();
        virtual void Update(Subject* theChangedSubject) = 0;
protected:
        Observer();
};
```

This implementation supports multiple subjects for each observer. Thesubject passed to the Update operation lets the observerdetermine which subject changed when it observes more than one.

Similarly, an abstract class defines the Subject interface:

```
class Subject {
public:
        virtual ~Subject();
        virtual void Attach(Observer*);
        virtual void Detach(Observer*);
        virtual void Notify();
protected:
        Subject();
private:
List<Observer*> *_observers;
};
```

```
void Subject::Attach (Observer* o) {      _observers->Append(o);    }

void Subject::Detach (Observer* o) {      _observers->Remove(o);    }

void Subject::Notify () {
ListIterator<Observer*> i(_observers);
for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Update(this);
}
}
```

ClockTimer is a concrete subject for storing andmaintaining the time of day. It notifies its observers every second.ClockTimer provides the interface for retrieving individualtime units such as the hour, minute, and second.

```
class ClockTimer : public Subject {
public:
        ClockTimer();
        virtual int GetHour();
        virtual int GetMinute();
        virtual int GetSecond();
        void Tick();
};
```

The Tick operation gets called by an internal timer atregular intervals to provide an accurate time base. Tickupdates the ClockTimer's internal state and callsNotify to inform observers of the change:

```
void ClockTimer::Tick () {
        // update internal time-keeping state
        // ...
        Notify();
}
```

Now we can define a class DigitalClock that displays thetime. It inherits its graphical functionality from a Widgetclass provided by a user interface toolkit. The Observer interface ismixed into the DigitalClock interface by inheriting fromObserver.

```
class DigitalClock: public Widget, public Observer {
public:
        DigitalClock(ClockTimer*);
        virtual ~DigitalClock();
        virtual void Update(Subject*);
                // overrides Observer operation
```

```
        virtual void Draw();
                // overrides Widget operation;
                // defines how to draw the digital clock
private:
        ClockTimer* _subject;
};

DigitalClock::DigitalClock (ClockTimer* s) {
        _subject = s;
        _subject->Attach(this);
}

DigitalClock:: DigitalClock () {
        _subject->Detach(this);
}
```

Before the Update operation draws the clock face, it checksto make sure the notifying subject is the clock's subject:

```
void DigitalClock::Update (Subject* theChangedSubject) {
        if (theChangedSubject == _subject) {
                Draw();
        }
}

void DigitalClock::Draw () {
        // get the new values from the subject

        int hour = _subject->GetHour();
        int minute = _subject->GetMinute();
        // etc.

        // draw the digital clock
}
```

An AnalogClock class can be defined in the same way.

```
class AnalogClock : public Widget, public Observer {
public:
        AnalogClock(ClockTimer*);
        virtual void Update(Subject*);
        virtual void Draw();
        // ...
```

```
};
```

The following code creates an AnalogClock and aDigitalClock that always show the same time:

```
ClockTimer* timer = new ClockTimer;
AnalogClock* analogClock = new AnalogClock(timer);
DigitalClock* digitalClock = new DigitalClock(timer);
```

Whenever the timer ticks, the two clocks will be updatedand will redisplay themselves appropriately.

Known Uses

The first and perhaps best-known example of the Observer pattern appearsin Smalltalk Model/View/Controller (MVC), the user interface framework in the Smalltalkenvironment [KP88]. MVC's Model class plays the role ofSubject, while View is the base class for observers. Smalltalk,ET++ [WGM88], and the THINK class library [Sym93b] provide ageneral dependency mechanism by putting Subject and Observer interfacesin the parent class for all other classes in the system.

Other user interface toolkits that employ this pattern areInterViews [LVC89], the AndrewToolkit [P+88], and Unidraw [VL90]. InterViewsdefines Observer and Observable (for subjects) classes explicitly.Andrew calls them "view" and "data object," respectively. Unidrawsplits graphical editor objects into View (for observers) and Subjectparts.

Related Patterns

Mediator (305): Byencapsulating complex update semantics, the ChangeManager acts asmediator between subjects and observers.

Singleton (144):The ChangeManager may use the Singleton pattern to make it uniqueand globally accessible.