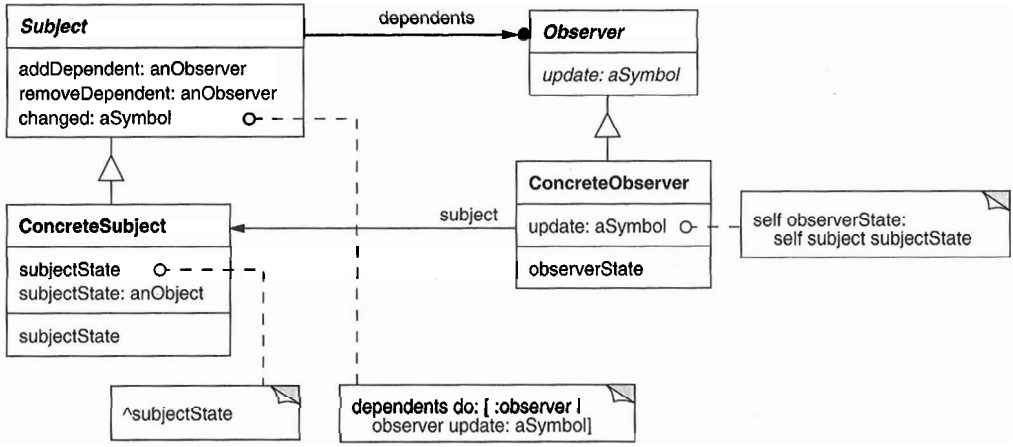
OBSERVER (DP 293) — Object Behavioral

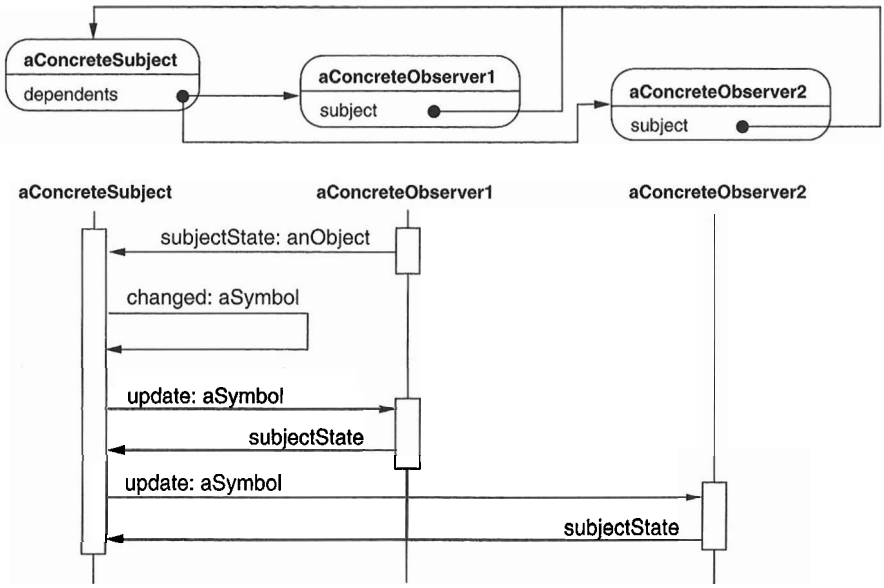
Intent

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

Structure



The interaction diagram on the following page illustrates the collaborations between a subject and two observers:



Discussion

Object-oriented techniques encourage the designer to break a problem apart into pieces—objects—that have a small set of responsibilities but can collaborate to accomplish complex tasks. This makes each object easier to implement and maintain, more reusable, and the combinations more flexible. The downside is that any simple behavior is so distributed across multiple objects that any change in one object often affects many others. All changes could be implemented to affect all relevant objects directly, but that would bind the objects together and ruin their flexibility, so objects need an easy, flexible way to tell each other about changes and events. This is the spirit of the Observer pattern.

The key to the Observer pattern is two objects, one of which holds the state that the other needs to share. We call them **Subject** and **Observer**, where a Subject holds the state and an Observer shares it. The Subject's collaboration with its Observer is very indirect so that the Observer may be optional. This way, the Subject can support multiple Observers just as easily as one or none at all. The Subject is unaware of whether it has any Observers, so it always announces changes in its state just in case. Each object that wishes to be an Observer registers its interest with the Subject so that it will receive these change announcements. While otherwise performing its duties, each Observer waits for notification from the Subject that its state has changed. We call it an Observer because of this way that it observes the Subject for state changes and keeps itself synchronized with its Subject. When it receives notification of a change, each Observer reacts in its own appropriate way, which can include ignoring irrelevant changes. Meanwhile, the Subject is unaware of what Observers it has or how various ones react

to particular changes. This observation relationship allows the Subject to concentrate on its own behavior and distributes the synchronization responsibilities to the Observers.

One confusing aspect of the Observer pattern in Smalltalk is that both the Subject and Observer protocols are implemented in one class, `Object`. This means that any object can serve as a Subject, an Observer, or both. Thus, when the pattern refers to a Subject or an Observer, remember that this is a distinction of the object's role, not its class. The object is not an instance of `Subject` or `Observer`; it is just an object.

Pattern or Language Feature?

As early as Smalltalk-80, Smalltalk had three kinds of relationships between objects (Goldberg & Robson, 1983, p. 240):

1. An object's references to other objects (i.e., its instance variables).
2. A relationship of a class to its superclass and metaclass (i.e., the instance side of a class knows what its class side and superclass are).
3. The ability of an object to register itself as a *dependent* of another object.

This third type, the dependency relationship used to "coordinate activities between different objects," is the Observer pattern. It is used to communicate between layers in an architecture (Brown, 1995a). Its best-known use is in the Model-View-Controller (MVC) framework for implementing the user interface in ParcPlace Smalltalk. MVC factors the interface into two layers of objects: a Model and a View-Controller pair. A guiding principle of MVC is that a Model should be able to support any number of View-Controller pairs, including none at all. Thus, a Model does not collaborate with its View-Controllers directly. It announces changes to its *dependents*—objects that have said they want to be notified of such changes. Each View and Controller registers itself as a dependent of the Model it is displaying. Then when the Model changes, the Views and Controllers get notified. MVC in Smalltalk became the standard from which all other graphical window systems were designed.

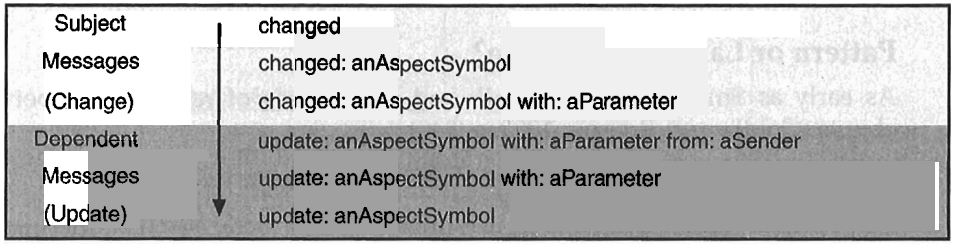
As fundamental as Observer appears to be in Smalltalk, it is not a language feature; it is a feature of most Smalltalk libraries. Every dialect implements it differently. The proposed ANSI Smalltalk standard doesn't include Observer at all (X3J20, 1996). Furthermore, window painters and other visual programming tools automate the use of Observer so that programmers often don't use it directly anymore. Nevertheless, Observer is a significant feature in Smalltalk that Smalltalk programmers should understand.

Observer in Smalltalk-80

The Observer pattern is implemented by the change/update protocol, introduced in Smalltalk-80 and still used in VisualWorks today (Woolf, 1994). The

protocol also exists in similar forms in Visual Smalltalk and IBM Smalltalk, but those dialects mostly use the event protocols described later.

The change/update protocol in Smalltalk-80 is implemented in `Object` as a combination of the Subject and Observer classes. This enables any object to be a Subject, an Observer (which Smalltalk calls a dependent), or both. When the Subject changes, it sends itself one of the change messages. The protocol converts this to a corresponding update message that is sent to each of the Subject's dependents. This is the series of messages:



The change/update protocol adds a key extension to the Observer pattern. When a subject announces that it has changed, it also specifies what has changed. When announcing a change, the subject specifies the update aspect that has changed. An *update aspect* is an object, usually a `Symbol`, that specifies a type of change that can occur in a subject. The subject defines the update aspects for its changes, and the dependents distinguish between those changes by checking the update aspect. If the update aspect is not one that the dependent is interested in, the dependent will ignore the change announcement.

A subject announces a change by sending itself one of the change messages:

- `changed`—The subject sends itself `changed` to announce that something has changed without specifying what has changed.
- `changed:`—The subject sends itself `changed:` to announce that a specific aspect has changed. The parameter, `anAspectSymbol`, is typically a `Symbol` and specifies the update aspect of the aspect that changed.
- `changed:with:`—The subject sends itself `changed:with:` to announce not only what specific aspect changed but also some extra information about the change. The second parameter, `aParameter`, is that extra information and can be any `Object`. It is often the new value of the aspect that changed.

A dependent receives a change notification by receiving the series of update messages. By default, a dependent will ignore all updates. A subclass that wishes to listen for any change notifications must implement one or more of the update messages to do so:

- `update:`—The dependent implements this message if it just needs to know what update aspect changed. The parameter, `anAspectSymbol`, is the update aspect that the subject specified when it made the change announcement. If the subject did not specify an update aspect, the parameter will be `nil`.
- `update:with:`—The dependent implements this message if it also needs to know the extra information that the subject specified about the change. The second parameter, `aParameter`, is the same object as the second parameter in `changed:with:`.
- `update:with:from:`—The dependent implements this message if it also needs to know which subject announced the change. The third parameter, `aSender`, is the subject that sent the change announcement.

Smalltalk-80: Creating Dependencies

A subject does not notify every other object in the image of its changes. That would be inefficient. Instead, an object that wants to receive certain notification registers itself with the object or objects that provide the desired notification. This way, only objects that have said they might want to know about a change are notified. When an object wants to receive notification from another object, it registers itself as a dependent on that object using the message `addDependent:`, as shown below. When it no longer wishes to receive notification, it removes itself as a dependent using the message `removeDependent:`, also shown below:

```
aSubject addDependent: aDependent.
...
aSubject removeDependent: aDependent.
```

A subject uses a collection to keep track of its dependents. In `Object`, this collection is a value in the `DependentsFields` dictionary. Only objects that have dependents have keys in this dictionary. `Model` tracks dependents more efficiently with its own `dependents` instance variable that bypasses `DependentsFields`. Thus, an object that is going to issue change notifications frequently will be more efficient if it is a subclass of `Model`.

The `addDependent:` and `removeDependent:` messages are often sent from the dependent's implementors of `initialize` and `release`, respectively. A dependent can register itself as a dependent on the same subject multiple times, creating redundant dependencies. However, redundant dependencies are inefficient and difficult to break because the dependent usually does not realize it has more than one dependency on the same subject. Thus, a dependent should be careful to register itself as a dependent on a subject only once. That way, a single send of `removeDependent:` will break the one and only dependency between the two objects.

Often dependent objects do not seem to be garbage collected properly. Over time, this will cause the development image to become bigger and slower for no

apparent reason. If these objects are no longer in use, why aren't they garbage collected? Because their dependency is still established and their Subject has not yet been garbage collected. A dependent will not be garbage collected until its Subject is. The class variable in `Object` used to track dependents for non-`Models` (e.g., `DependentsFields` or `Dependents`) can cause similar problems even after the Subject is no longer being used. So when a dependent object will no longer be used, it should release its dependencies so that it can be garbage collected.

Observer in VisualWorks

Since Smalltalk-80 is the foundation of VisualWorks, VisualWorks includes the change/update protocol described above. When VisualWorks introduced the window painter, it also added some new features to its implementation of the Observer pattern.

VisualWorks: DependencyTransformer

With change/update, the dependency relationships between objects are often difficult to follow. A framework might contain several subjects that are issuing numerous change notifications and several dependents that are listening for numerous change notifications. The cause and effect between any one change and its corresponding response is difficult to see, especially in static code. Threads of control that are difficult to follow cannot be maintained well. Thus, change/update needs a better way to implement the relationship between a change in the subject and a corresponding reaction in a dependent.

VisualWorks clarifies this relationship with a class called `DependencyTransformer`. A `DependencyTransformer` is an Adapter (105) for a dependent that converts the subject's generic `update:` message to a more specific message in the dependent's interface. It acts as a bridge between a change in a subject and a corresponding reaction in a dependent. It encapsulates three items together: the dependent, what aspect it's interested in, and what message to invoke when this aspect changes. This is an example of the Self-Addressed, Stamped Envelope (SASE) pattern discussed below.

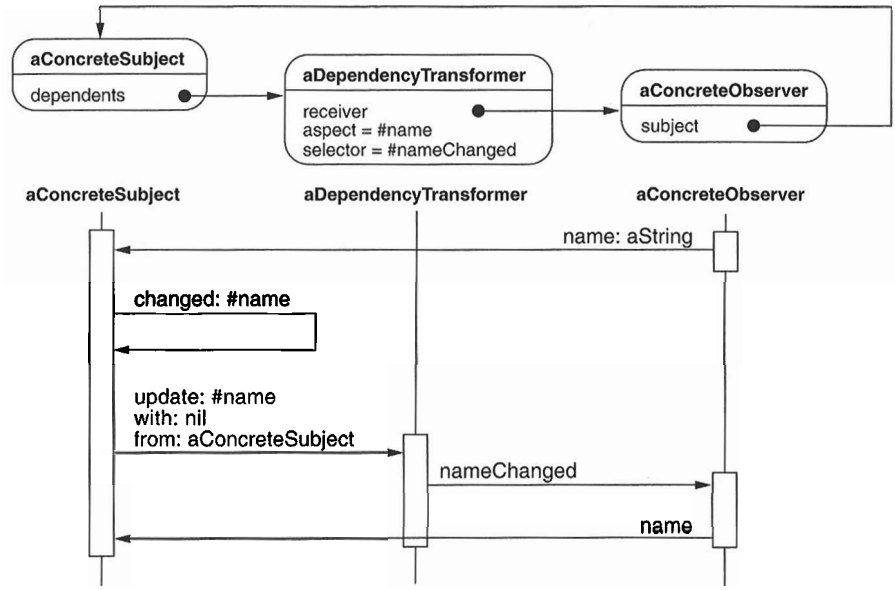
Since any object can be a subject, any object can have a `DependencyTransformer`. As shown below, the message for creating a `DependencyTransformer` is `expressInterestIn:for:sendBack:` and for breaking the dependency is `retractInterestIn:for:`. The two messages use `addDependent:` and `removeDependent:` to make the `DependencyTransformer` a dependent of the subject:

```
aSubject
   expressInterestIn: anAspectSymbol
   for: aDependent
   sendBack: aSelector.
...
aSubject
   retractInterestIn: anAspectSymbol
   for: aDependent.
```

Like `addDependent:` and `removeDependent:`, a dependent often sends these `DependencyTransformer` messages from `initialize` and `release`.

The code and diagram below show how a `DependencyTransformer` is created, how it sits between the subject and its observer, and the message interaction when the `DependencyTransformer` fields an update and notifies the observer:

```
ConcreteObserver>>initialize
    ...
    aConcreteSubject
        expressInterestIn: #name
        for: self
        sendBack: #nameChanged.
    ...
```



VisualWorks: ValueModel

Another shortcoming of change/update is that every dependent of a subject is notified of every change in that subject. A change is usually a change in the value of an aspect, which essentially means an instance variable's value has changed. A subject often contains more aspects than a dependent is interested in. Thus, the dependent wastes time receiving notification of changes in aspects it's not interested in. This can even lead to subtle bugs when the dependent misinterprets which aspect changed.

VisualWorks uses `ValueModels` to encapsulate an aspect and its value as a first-class object. Some languages call this an "active variable." Unlike a plain variable, a `ValueModel` is itself an object that can be inspected, sent messages, and

```
AspectAdaptor>>update: anAspect with: parameter from: sender
   "Propagate change if the sender is the receiver's subject
   and anAspect is the receiver's aspect."
   (sender == subject and: [anAspect == self forAspect])
      ifTrue:
         [dependents
            update: #value with: parameter from: self]
      ifFalse:
         [super
            update: anAspect with: parameter from: sender]
```

Notice that we had to implement the Subject behavior in `PhoneNumber`. On the other hand, `AspectAdaptor` already implemented the Observer behavior for us. This is typical in VisualWorks. We have to implement the Subject ourselves, but VisualWorks has already implemented the Observers.

Observer in Model-View-Controller

The Model-View-Controller framework is the first and best-known example of Observer. Introduced in Smalltalk-80, it lives on in VisualWorks. The main examples of Observer exist between the `ValueModels` and their value-based subviews (Woolf, 1995a). For example, the subview classes for an input field and the field part of a combo-box are `InputFieldView` and its subclass `ComboBoxInputFieldView`.

When a `ValueModel` gets a new value, it announces this to its dependents by sending itself `changed:`.

```
ValueModel>>value: newValue
   "Set the currently stored value, and notify dependents."
   self setValue: newValue.
   self changed: #value
```

When a subview gets notified of a change, it gets its model's new value and displays it. The subview listens for changes in the model by registering itself as a dependent using `addDependent:`. This is implemented in `DependentPart`, the superclass for all views that can have models (such as `View`):

```
DependentPart>>setModel: aModel
   "Set the receiver's model to be aModel."
   ( model == aModel )
      ifFalse: [
         model isNil
            ifFalse: [ model removeDependent: self ]
         (model := aModel) isNil
            ifFalse: [ model addDependent: self ] ].
```

Since a value-based subview has only one model and the model has only one aspect, the subview doesn't even check to see what change is being announced.

The subview assumes that the model's value is what changed and proceeds to get the new value. The subview listens for change announcements by implementing `update:` to do so:

```
InputFieldView>>update: aSymbol
   "The receiver's model has changed its text.
   Update the receiver."
   self updateDisplayContents
```

In this example, the `ValueModel` is the Subject and the subview is the Observer. Notice that this `ValueModel` is in an `ApplicationModel` and may well be both an Observer on a domain model and a Subject for the subview.

Since VisualWorks already implements both the `ValueModel` classes and the subview classes like `InputFieldView`, you do not have to implement any code yourself to use this example of Observer.

Observer in Visual Smalltalk and IBM Smalltalk

For sample code of Observer in Visual Smalltalk and IBM Smalltalk, see the sample code in Mediator (287).

Known Smalltalk Uses

Each dialect implements one or two variations of the changed/update protocol from Smalltalk-80, but these implementations are so fundamental to the language that they're the only ones you need. Most Smalltalk applications contain numerous uses of the changed/update protocol, and each of these is a known use of Observer. Uses of the SASE variation of Observer are also extremely common. The Observer pattern is so fundamental to Smalltalk that it is almost a language feature.

Related Patterns

Many of the classes in the `ValueModel` hierarchy in VisualWorks are Observer classes. Of those, some are also Adapter (105) classes, so their Subject is also the Adaptee. The others are Decorator (161) classes, so their Subject is also the Component.

An application model, an Observer of its domain model, is also a Mediator (287) for its widgets.

A value-based subview, an Observer of its `ValueModel`, also uses its `ValueModel` as a Strategy (339) for accessing its value. The `ValueModel` also uses the subview as a Strategy for displaying its value.