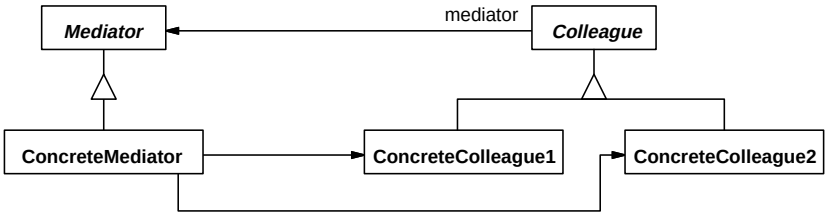
MEDIATOR (DP 273) Object Behavioral

Intent

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Structure



Discussion

One of the characteristics of object technology is that it helps designers manage complexity better than traditional design methods. Good object-oriented design increases cohesion within objects and decreases coupling between them. It composes a system of collaborating objects, each with distinct responsibilities, whose configuration dictates how they will work together. The problem is: Where's the behavior? It's not in one object or another, it's between the objects. This is very flexible when each object has a few interactions with a couple of collaborators. But it can lead to designs where the objects are so highly dependent on each other that changes in any one can affect all of the others. When this happens, each object manages itself well, but managing these complex connections becomes a problem. Mediator can solve this problem.

The key to the Mediator pattern is a set of objects that need to collaborate but shouldn't know about each other directly, so instead they all collaborate with a central object that keeps them coordinated. The coordinating object, called a Mediator, calls the objects it's coordinating its Colleagues. Events in one Colleague potentially affect many others, but for all of them to collaborate together directly would become exponentially complex. Their design would become dependent on each other such that removing any one of them would require redesigning the others. Instead, all of the Colleagues collaborate with the Mediator, which in turn collaborates with the Colleagues. If one Colleague is removed from the design, the Mediator is redesigned accordingly and the other Colleagues remain unchanged.

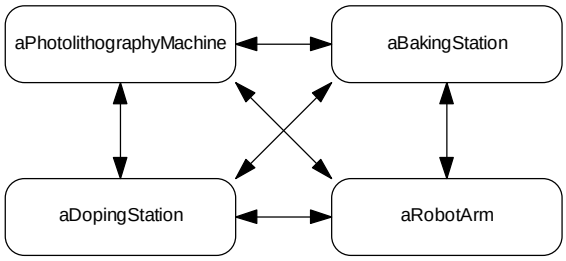
Many problem domains involve complex connections between the various objects of the system. Handling these complex interactions is difficult. For example, let's consider a semiconductor manufacturing system.

Semiconductor Manufacturing

On a factory floor, many different machines work on semiconductor wafers in various stages of production. The machines are connected in an assembly line so that wafers pass from one stage of production to the next. Semiconductor manufacturing is a complex process. Some of the steps in the process must occur within very fine time tolerances. If these tolerances are not met, then thousands of dollars worth of materials are lost.

What happens when something goes wrong at a machine? A photolithography machine's alignment sensors can drift out of spec, or a doping station might run out of chemicals, or a baking station's ovens could fall below the minimum acceptable temperature. If this happens, the other machines in the line must adjust their operations, perhaps even shutting themselves down completely, but in an orderly fashion.

One potential solution is to interconnect all of the machines in the assembly line together, as shown in the following diagram. That way, if one raises an alarm, it will notify all of the rest.

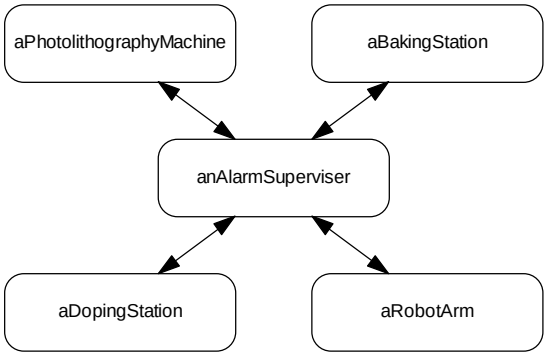


However, this approach presents a few problems:

- Some of the machines are not affected by certain alarms. Yet precious time is spent notifying every machine about every alarm. Worse yet, the machines that need to respond first may not be notified in time.
- Knowledge of how the machines connect is spread and duplicated throughout the system. If a new machine is added, or the manufacturing process changes, then code spread throughout the system must be changed.

Another solution is to introduce a new object, an Alarm Supervisor, as shown below. This object's responsibility is to listen for and respond to alarms. It notifies only those machines that are affected, and does so in the optimal order. This way, each machine

can concentrate on performing its own tasks independently of the others. It need only inform the Alarm Sensor, and only when one of its alarms goes off.



The heart of the Mediator pattern is this approach of taking a number of complex relationships between objects and reifying them into another, distinct object. Mediator avoids the problems of complex interconnection by encapsulating the connection behavior into a separate Mediator object. It is responsible for coordinating the interactions of the objects in the group. Each object in that group, a Colleague, knows only about the Mediator, not about the other members in the group. The Colleagues send requests to the Mediator, who notifies the other Colleagues as necessary.

A Warning About Mediator

Mediator is often abused. Misapplication of the pattern "…can make the mediator itself a monolith that's hard to maintain." (DP 277) Novice OO designers often take the intent of the mediator too literally, trying to encapsulate every set of object interactions. They use the pattern to justify a poor factoring of responsibilities.

Many novice designers create designs containing a large number of "data holders," objects that have little behavior other than getter and setter methods. They then find that these objects are too simple to perform the system's work. To remedy this, the designer creates a set of "Manager" objects that contain all of the functionality of the system, and that operate on the "data holders." To call such objects "mediators" is a misuse of the term.

The purpose of a Mediator is to manage the relationships between numerous objects so that they can each focus on their own behavior independently of the others. A Manager object, on the other hand, tends to extract state out of one object, manipulate it, and plug the result into another object. The objects do not encapsulate their own behavior; the Manager does that for them and makes them passive. You should strive to make your objects responsible for their own behavior and to coordinate with each other as neces-

sary. Then you will not need a Manager object. If the objects' coordination becomes overly complex, introduce a Mediator to manage their coordination but not to manage their fundamental behavior.

Sample Code

A layered architecture is a common architectural choice in many applications. Understanding the roles of different objects in an application, and the way control and information flow between them is crucial to having a high-level understanding of a complex application. Brown (1996a) describes a layered architecture for Smalltalk applications that consists of four layers. The layers are (in order from the top) the View layer, the Application Model layer, the Domain layer, and the Infrastructure layer. Buschmann et. al. (1997) describe a similar architecture for IS systems that parallels this division. To understand why this division of roles is necessary, we have to examine the evolution of the MVC application framework.

Application Model and Domain Model

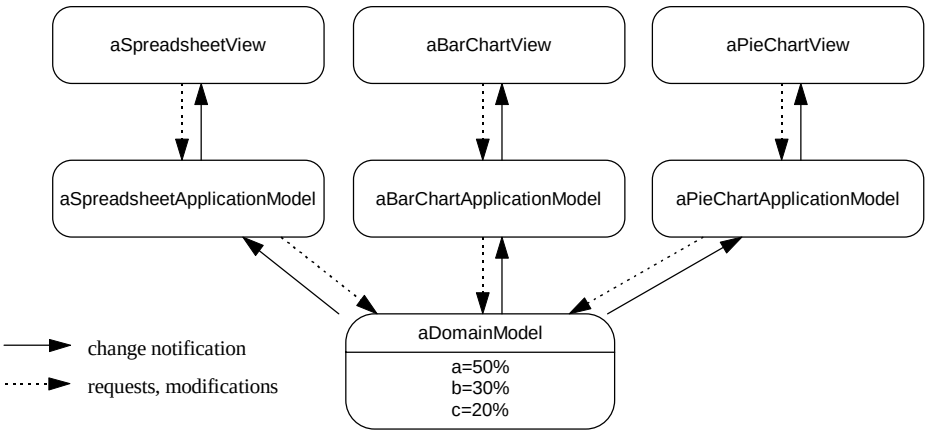
In the original Model-View-Controller (MVC) framework, Models have two roles. The Model is responsible for representing a domain object and storing its state, and is also responsible for supporting the View's display of that state. The early examples of MVC (Krasner & Pope, 1988) showed their Model objects both opening Views on themselves and coordinating their display, as well as handling domain-specific behavior. Over time, developers began to realize that this was a lot of responsibility for one object. As Models evolved to support several different types of Views, they became increasingly complex and difficult to implement and maintain. As a result, the MVC framework evolved to factor the Model into two parts, the Domain Model and the Application Model (Woolf, 1995).

The separate Domain and Application Models simplify the Model concept. Each domain model (also called a domain object) represents an object in the domain. These are the business objects, the classes in the object model. They know nothing about the user interface or how it will display them, if at all. They work just fine even if there is no user interface.

The application model provides the user interface support. The View still does the display, but the application model provides the support it needs. An application model does not contain much state. Instead, it derives its state from one or more domain models. As Brown (1996a) demonstrates, the Application Model fulfills two roles. It acts as an Adapter to convert the domain's interface into the interface that the View expects. In the process, the application model provides resources that are not part of the domain, such as menus. It also acts as a Mediator to coordinate the widgets in the View.

As an example, look at the user interface example diagram in the Motivation for Observer (DP 293). It shows a set of data and three different views of that data: a spreadsheet, a bar chart, and a pie chart. In this enhanced MVC framework, the objects would

be a domain object that contains the data, three application models to display the data, and one window for each of the application models.

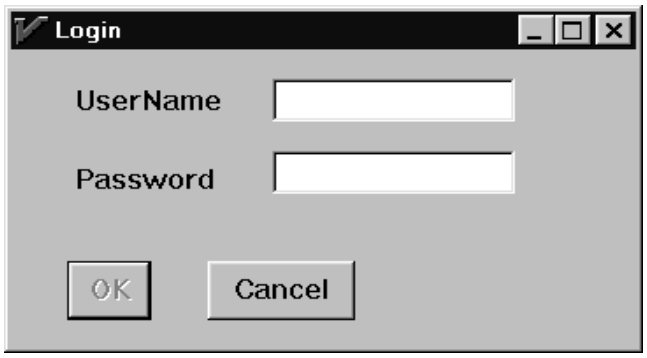


Notice that the above diagram shows that each different View would have its own specific Application Model. In actuality, each Window in a user interface would have an underlying application model that coordinated multiple views. This coordination of Views is what makes the Mediator pattern so crucial to understanding the Smalltalk window system architectures.

Design Patterns mentions that Visual Smalltalk uses the Mediator pattern in its application architecture (page 281). In fact, all three major dialects of Smalltalk utilize it in nearly the same way. Let's look at how their implementations differ by examining a common example in all three dialects.

Our example is a simple Login dialog, shown in the figure below, used to obtain a user name and password for a mainframe or relational database (Hendley & Smith, 1992). There are three types of collaborations between UI widgets in an application model:

- simple actions that only affect one UI widget and not the domain model;
- complex actions that affect multiple UI widgets; and
- simple actions that affect both the UI and domain model.



The three major dialects handle these collaborations differently, as this example will show. In our example, we will look at three different interactions that occur within the dialog:

- The Password Field is disabled until the user begins typing in the User Name field.
- The OK button is disabled until the user has entered something into both the User Name and Password fields.
- The OK button causes the domain model to verify the user name and password. If the pair is valid, the window closes. Otherwise, the window displays a message indicating that the user name/password pair entered is not valid.

Mediator in Visual Smalltalk

Design Patterns describes the Visual Smalltalk application architecture in detail (page 281). It discusses the event mechanism used to communicate between the panes in a window and their Mediator, a `ViewManager`. Our Observer discussion documents this mechanism as the SASE variation.

`ViewManager` is part of the older Smalltalk/V application architecture, and is maintained for backward compatibility. Applications may now alternatively use a new Mediator class, `ApplicationCoordinator`. The differences between `ApplicationCoordinator` and `ViewManager` are minor, and not relevant to this discussion. For more information, see the *Visual Smalltalk User's Guide*.

The most interesting interaction in the example is the requirement that the OK button be disabled until the user has entered something into both the user name and password fields. To accomplish this, we have to check that there is something in both fields before we enable the button. The easiest way to do this is to connect the `#textChanged` event on both entry fields to a method in the `ViewManager` that looks at the contents of both before enabling the button. To see how this is accomplished, let's look at the following code snippet from the method in `LoginViewManager` that sets up the panes.

```
LoginViewManager>>someMethodName
    ...
    userNameEF "an EntryField"
        owner: aModel;
        setName: 'userNameEF';
        when: #textChanged: send: #tryToEnableOK to: self.
    passwordEF "an EntryField"
        owner: aModel;
        setName: 'passwordEF';
        when: #textChanged: send: #tryToEnableOK to: self.
    ...
```

As described in Observer, the message

```
when: anEvent send: aSelector to: aReceiver
```

creates a `Message` object. When `anEvent` occurs in the pane, the message whose selector is `aSelector` is sent to `aReceiver`. In this case, the event is each field's text changing, the message is `tryToEnableOK`, and the receiver is the `LoginViewManager`.

This code shows how `tryToEnableOK` works:

```
LoginViewManager>>tryToEnableOK
    | test1 test2 |
    test1 := (self paneNamed: 'userNameEF') contents notEmpty.
    test2 := (self paneNamed: 'passwordEF') contents notEmpty.
     (test1 and: [test2])
        ifTrue: [(self paneNamed: 'okPB') enable]
```

In this case, the Mediator pattern is critical. The `ViewManager` does not need to handle all of the user interface actions in the Visual Smalltalk framework. The receiver of the SASE can just as easily be another pane. In fact, it is sometimes easier to set up such connections in very simple cases. For example, if enabling the OK button depended on *just* the user name field, we could have set up the field's text changed event to *directly* enable the OK button:

```
    userNameEF "an EntryField"
        owner: aModel;
        setName: 'userNameEF';
        when: #textChanged: send: #enable to: okPushButton
```

However, when an interaction requires that two or more panes be involved, sending messages to the `ViewManager` is the cleanest solution. This is the Mediator pattern. When we look at the implementation of the Mediator pattern in IBM Smalltalk and VisualAge for Smalltalk, this becomes even more apparent.

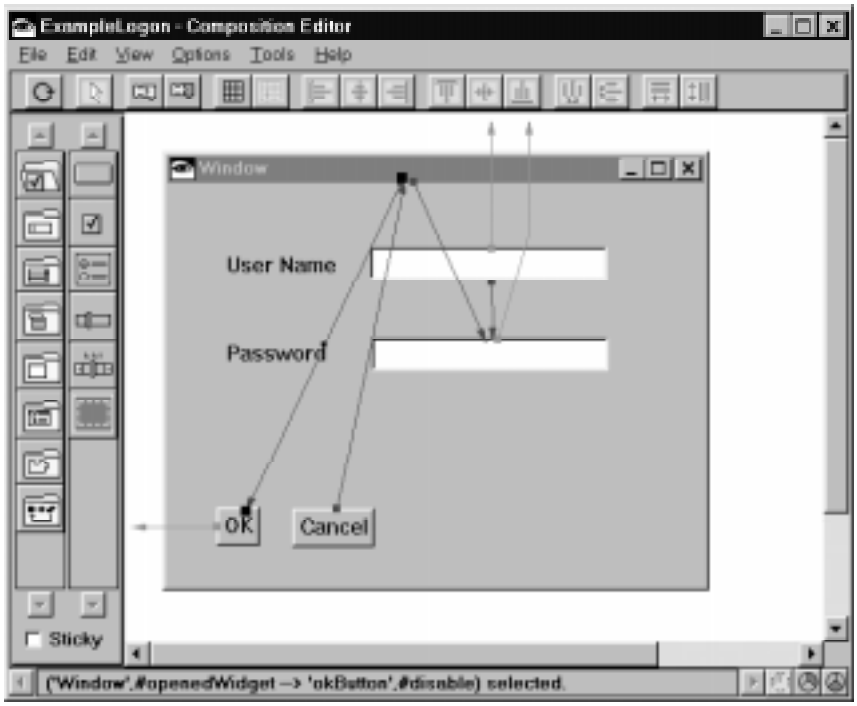
Mediator in IBM Smalltalk and VisualAge for Smalltalk

Unlike the other Smalltalk dialects, IBM Smalltalk does not supply an abstract application model class that serves as a Mediator. IBM Smalltalk's windowing system is based on OSF Motif, and Motif does not use Mediator. Instead, to implement a Mediator, you subclass it directly off of `Object`. This is a valid implementation choice documented in *Design Patterns* (page 278).

VisualAge for Smalltalk, a visual programming environment built on top of IBM Smalltalk, adds an abstract application model class, `AbtAppBuilderView`. Windows built in VisualAge are subclassed from that class. However, VisualAge application models do not really incorporate the Mediator pattern directly. Instead, VisualAge encourages direct connections between UI widgets.

As an example, let's look at a diagram of the Login window as implemented in VisualAge. Below is an example of the VisualAge Composition Editor. It has four types of connections that can be made between objects ("parts") on the screen:

- *Attribute-to-Attribute* — These connections link data items directly.
- *Event-to-Action* — These connections link events to actions (predefined methods) of some object.
- *Event-to-Script* — This means that the occurrence of an event triggers the running of a script (a user defined method).
- *Attribute-to-Script* — In this case, a script executes to calculate a value for the attribute; this is a form of lazy initialization.



IBM Smalltalk and VisualAge for Smalltalk use the same event mechanism as Visual Smalltalk. Events trigger corresponding methods that react to the event. This is the SASE variation of Observer. All four types of connections in VisualAge use SASE (through callbacks) to send a `DirectedMessage` from one part to another when an event occurs.

Attribute-to-Attribute and Event-to-Action connections link two parts together directly. In terms of the Mediator pattern, this is communication between Colleagues. The other two types of connections, Event-to-Script and Attribute-to-Script, are links between the parts and the Mediator (or "visual part") that the user is building. An Event-to-Script connection is an instance of a Colleague informing the Mediator of an event. Each Attribute-to-Script connection (which should actually be called "Script-to-Attribute") is an instance of the Mediator communicating with a Colleague, presumably in response to an event.

Therefore, an Event-to-Script connection is the true example of Mediator. While another form of connection may be visually cleaner and simpler to use, it will not be as easily extendible.

For example, to implement the login example in VisualAge, we create two Event-to-Script connections. They go from each entry field to the script named `tryToEnableOK`. This is the code for that script:

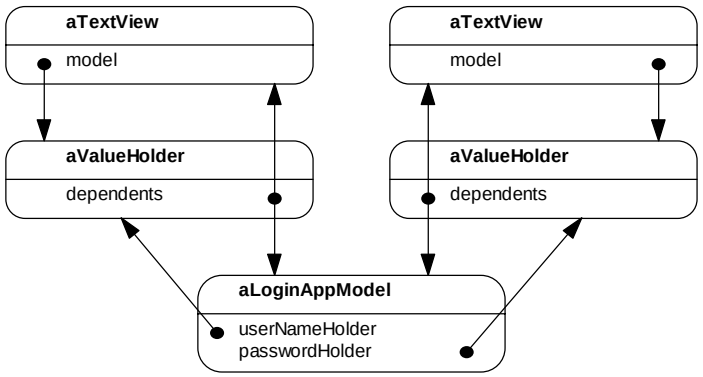
```
tryToEnableOK
    | test1 test2 |
    test1 := (self subpartNamed: 'userName') string notEmpty.
    test2 := (self subpartNamed: 'password') string notEmpty.
    (test1 & test2)
        ifTrue: [(self subpartNamed: 'okButton') enable]
```

As you can see, the VisualAge implementation is very similar to the Visual Smalltalk implementation.

Mediator in VisualWorks

As mentioned in Observer, VisualWorks uses a `ValueModel` as an intermediate object that reduces the total overhead of change notifications to dependents. The `ValueModel` sits between a View and an application model and encapsulates an aspect in the application model. The View is an Observer of the `ValueModel`. The application model may also observe the `ValueModel` using a `DependencyTransformer`. This additional layer of objects even further decouples the Views from the domain models.

The figure below shows the connections between the objects in our example as implemented in VisualWorks. The Button has been left out for simplicity, but it is also connected to the `LoginAppModel` through a `ValueModel`, a `PluggableAdaptor`.



This code shows the class definition for the example in VisualWorks. The VisualWorks UI Builder automatically adds the instance variables `userNameHolder` and

passwordHolder. The UI Builder automatically adds lazy initialization code to set these instance variables to hold a ValueHolder on an empty String.

```
SimpleDialog subclass: #LoginAppModel
    instanceVariableNames: 'userNameHolder passwordHolder'
    classVariableNames: ''
    poolDictionaries: ''
```

The connections between the two ValueHolders and LoginAppModel are made in initialize, as shown below. This method is automatically called when the LoginAppModel is created. The method onChangeSend:to: will create a DependencyTransformer that will send the message specified by the onChangeSend: parameter whenever the value of that ValueHolder changes.

```
LoginAppModel>>initialize
    "See superimplementor."
    super initialize.
    self userNameHolder onChangeSend: #tryToEnableOK to: self.
    self passwordHolder onChangeSend: #tryToEnableOK to: self
```

Now that the ValueHolders are set up, the message tryToEnableOK is run whenever the value of either ValueHolder changes. The method is shown below. As you can see, its implementation is similar to the two previous implementations.

```
LoginAppModel>>tryToEnableButton
    | button test1 test2 |
    button := self builder componentAt: #okButton.
    button isNil ifTrue: [^self].
    test1 := self userNameHolder value notEmpty.
    test2 := self passwordHolder value notEmpty.
    (test1 and: [test2]) ifTrue: [button enable]
```

Known Smalltalk Uses

While Mediator is mostly commonly seen in Smalltalk designs in the application frameworks described earlier, its use is by no means restricted to user interface programming. Brown et al. (1994) describe a class named HostTransaction in the design of a framework for mainframe communications. HostTransaction acts as a mediator between HostScreen objects, which represent individual mainframe terminal screens in a transaction. The HostTransaction coordinates the navigation between the HostScreen objects and controls the flow of information between them.

The semiconductor manufacturing example discussed in the first section is drawn from a prototype system developed by KSC for a semiconductor equipment manufacturing company. It is similar to code used in the WORKS system developed by Texas Instruments for semiconductor fabrication facility automation.