Mediator

Intent

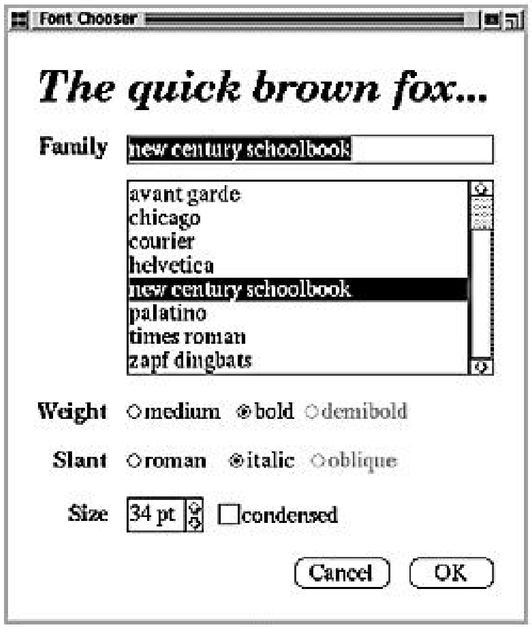
Define an object that encapsulates how a set of objects interact.Mediator promotes loose coupling by keeping objects from referring toeach other explicitly, and it lets you vary their interactionindependently.

Motivation

Object-oriented design encourages the distribution of behavioramong objects. Such distribution can result in an object structurewith many connections between objects; in the worst case, every objectends up knowing about every other.

Though partitioning a system into many objects generally enhancesreusability, proliferating interconnections tend to reduce it again.Lots of interconnections make it less likely that an object can workwithout the support of others—the system acts as though it weremonolithic. Moreover, it can be difficult to change the system'sbehavior in any significant way, since behavior is distributed amongmany objects. As a result, you may be forced to define many subclassesto customize the system's behavior.

As an example, consider the implementation of dialog boxes in agraphical user interface. A dialog box uses a window to present acollection of widgets such as buttons, menus, and entry fields, asshown here:

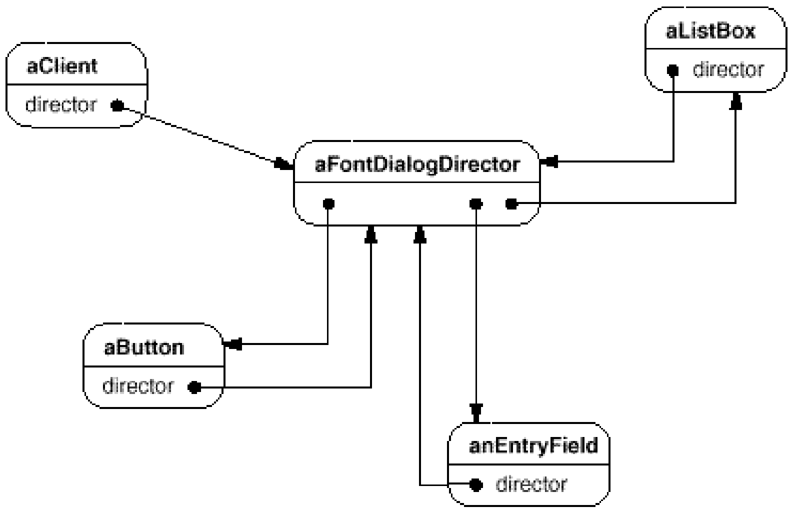


Often there are dependencies between the widgets in the dialog. Forexample, a button gets disabled when a certain entry field is empty.Selecting an entry in a list of choices called a **list box**might change the contents of an entry field. Conversely, typing textinto the entry field might automatically select one or morecorresponding entries in the list box. Once text appears in the entryfield, other buttons may become enabled that let the user do somethingwith the text, such as changing or deleting the thing to which it refers.

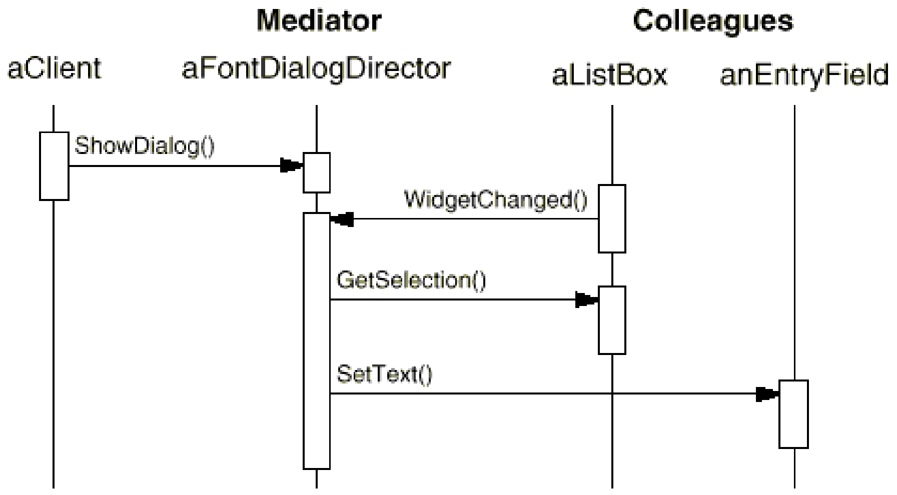
Different dialog boxes will have different dependencies betweenwidgets. So even though dialogs display the same kinds of widgets,they can't simply reuse stock widget classes; they have to becustomized to reflect dialog-specific dependencies. Customizing themindividually by subclassing will be tedious, since many classes areinvolved.

You can avoid these problems by encapsulating collective behavior in aseparate **mediator** object. A mediator is responsible forcontrolling and coordinating the interactions of a group of objects.The mediator serves as an intermediary that keeps objects in the groupfrom referring to each other explicitly. The objects only know themediator, thereby reducing the number of interconnections.

For example, **FontDialogDirector** can be the mediatorbetween the widgets in a dialog box. A FontDialogDirector object knowsthe widgets in a dialog and coordinates their interaction. It acts asa hub of communication for widgets:



The following interaction diagram illustrates how the objects cooperate tohandle a change in a list box's selection:

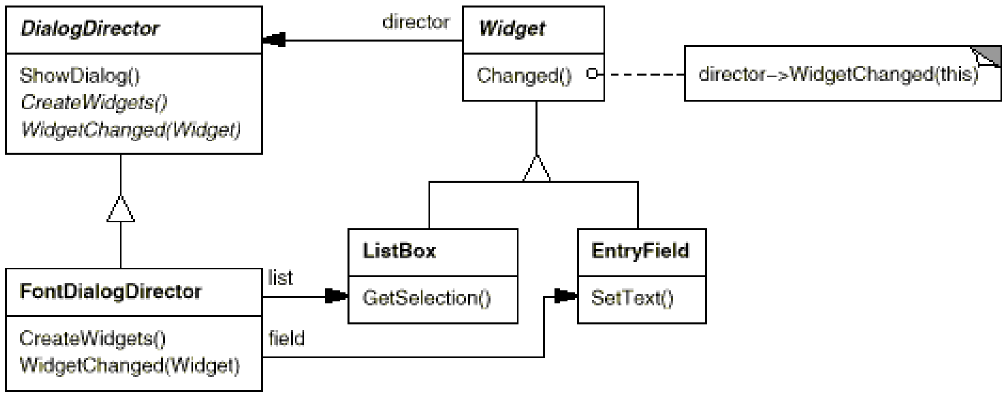


Here's the succession of events by which a list box's selection passesto an entry field:

1. The list box tells its director that it's changed.
2. The director gets the selection from the list box.
3. The director passes the selection to the entry field.
4. Now that the entry field contains some text, the directorenables button(s) for initiating an action (e.g., "demibold," "oblique").

Note how the director mediates between the list box and the entry field.Widgets communicate with each other only indirectly, through thedirector. They don't have to know about each other; all they know is thedirector. Furthermore, because the behavior is localized in one class,it can be changed or replaced by extending or replacing that class.

Here's how the FontDialogDirector abstraction can be integrated into aclass library:



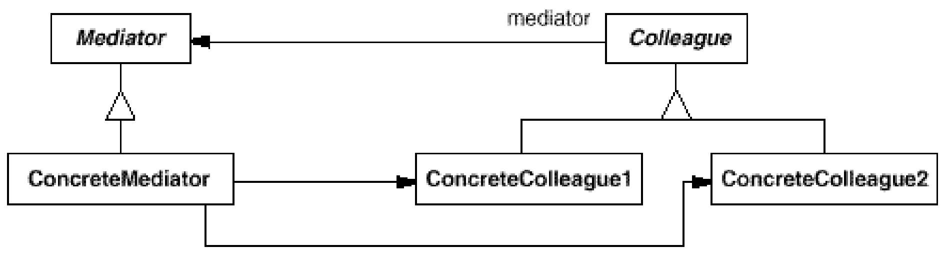
DialogDirector is an abstract class that defines the overall behavior ofa dialog. Clients call the ShowDialog operation to display the dialog onthe screen. CreateWidgets is an abstract operation for creating thewidgets of a dialog. WidgetChanged is another abstract operation;widgets call it to inform their director that they have changed.DialogDirector subclasses override CreateWidgets to create the properwidgets, and they override WidgetChanged to handle the changes.

Applicability

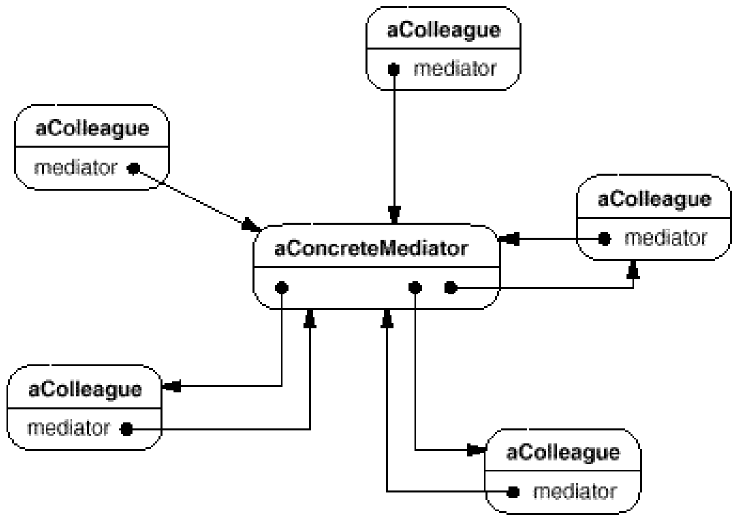
Use the Mediator pattern when

- a set of objects communicate in well-defined but complex ways. Theresulting interdependencies are unstructured and difficult tounderstand.
- reusing an object is difficult because it refers to and communicateswith many other objects.
- a behavior that's distributed between several classes should becustomizable without a lot of subclassing.

Structure



A typical object structure might look like this:



Participants

- **Mediator** (DialogDirector)
  - defines an interface for communicating with Colleague objects.
- **ConcreteMediator** (FontDialogDirector)
  - implements cooperative behavior by coordinating Colleague objects.
  - knows and maintains its colleagues.
- **Colleague classes** (ListBox, EntryField)
  - each Colleague class knows its Mediator object.
  - each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague.

Collaborations

- Colleagues send and receive requests from a Mediator object. Themediator implements the cooperative behavior by routing requestsbetween the appropriate colleague(s).

Consequences

The Mediator pattern has the following benefits and drawbacks:

1. *It limits subclassing.* A mediator localizes behavior that otherwise would be distributed amongseveral objects. Changing this behavior requires subclassing Mediatoronly; Colleague classes can be reused as is.
2. *It decouples colleagues.* A mediator promotes loose coupling between colleagues. You can varyand reuse Colleague and Mediator classes independently.
3. *It simplifies object protocols.* A mediator replaces many-to-many interactions with one-to-manyinteractions between the mediator and its colleagues. One-to-manyrelationships are easier to understand, maintain, and extend.
4. *It abstracts how objects cooperate.* Making mediation an independent concept and encapsulating it in anobject lets you focus on how objects interact apart from theirindividual behavior. That can help clarify how objects interact in asystem.
5. *It centralizes control.* The Mediator pattern trades complexity of interaction for complexity inthe mediator. Because a mediator encapsulates protocols, it can becomemore complex than any individual colleague. This can make the mediatoritself a monolith that's hard to maintain.

Implementation

The following implementation issues are relevant to the Mediatorpattern:

1. *Omitting the abstract Mediator class.* There's no need to define an abstract Mediator class when colleagueswork with only one mediator. The abstract coupling that theMediator class provides lets colleagues work with different Mediatorsubclasses, and vice versa.
2. *Colleague-Mediator communication.* Colleagues have to communicate with their mediator when an event ofinterest occurs. One approach is to implement the Mediator as anObserver using the Observer (326) pattern. Colleagueclasses act as Subjects, sending notifications to the

mediatorwhenever they change state. The mediator responds by propagating theeffects of the change to other colleagues.

Another approach defines a specialized notification interface inMediator that lets colleagues be more direct in their communication.Smalltalk/V for Windows uses a form of delegation: When communicatingwith the mediator, a colleague passes itself as an argument, allowingthe mediator to identify the sender. The Sample Code uses thisapproach, and the Smalltalk/V implementation is discussed further inthe Known Uses.

Sample Code

We'll use a DialogDirector to implement the font dialog box shown inthe Motivation. The abstract class DialogDirector definesthe interface for directors.

```
class DialogDirector {
public:
virtual ~DialogDirector();
virtual void ShowDialog();
virtual void WidgetChanged(Widget*) = 0;
protected:
DialogDirector();
virtual void CreateWidgets() = 0;
};
```

Widget is the abstract base class for widgets. Awidget knows its director.

```
class Widget {
public:
Widget(DialogDirector*);
virtual void Changed();
virtual void HandleMouse(MouseEvent& event);
// ...
private:
DialogDirector* _director;
};
```

Changed calls the director's WidgetChangedoperation. Widgets call WidgetChanged on their director toinform it of a significant event.

```
void Widget::Changed ()
{       _director->WidgetChanged(this);   }
```

Subclasses of DialogDirector overrideWidgetChanged to affect the appropriate widgets. The widgetpasses a reference to itself as an argument to WidgetChangedto let the director identify the widget that changed.DialogDirector subclasses redefine theCreateWidgets pure virtual to construct the widgets in thedialog.

The ListBox, EntryField, and Button aresubclasses of Widget for specialized user interfaceelements. ListBox provides a GetSelectionoperation to get the current selection, and EntryField'sSetText operation puts new text into the field.

```
class ListBox : public Widget {
public:
ListBox(DialogDirector*);
virtual const char* GetSelection();
virtual void SetList(List<char*>* listItems);
virtual void HandleMouse(MouseEvent& event);
// ...
};

class EntryField : public Widget {
public:
EntryField(DialogDirector*);
virtual void SetText(const char* text);
virtual const char* GetText();
virtual void HandleMouse(MouseEvent& event);
// ...
};
```

Button is a simple widget that calls Changedwhenever it's pressed. This gets done in its implementation ofHandleMouse:

```
class Button : public Widget {
public:
Button(DialogDirector*);
virtual void SetText(const char* text);
virtual void HandleMouse(MouseEvent& event);
// ...
};

void Button::HandleMouse (MouseEvent& event) {
// ...
Changed();
}
```

The FontDialogDirector class mediates between widgets in thedialog box. FontDialogDirector is a subclass ofDialogDirector:

```
class FontDialogDirector : public DialogDirector {
public:
FontDialogDirector();
virtual ~FontDialogDirector();
virtual void WidgetChanged(Widget*);
protected:
virtual void CreateWidgets();
private:
Button* _ok;
Button* _cancel;
ListBox* _fontList;
EntryField* _fontName;
};
```

FontDialogDirector keeps track of the widgets it displays. ItredefinesCreateWidgets to create the widgets and initialize itsreferences to them:

```
void FontDialogDirector::CreateWidgets () {
_ok = new Button(this);
_cancel = new Button(this);
_fontList = new ListBox(this);
_fontName = new EntryField(this);
// fill the listBox with the available font names
// assemble the widgets in the dialog
}
```

WidgetChanged ensures that the widgets work together properly:

```
void FontDialogDirector::WidgetChanged ( Widget* theChangedWidget ) {
if (theChangedWidget == _fontList) {
_fontName->SetText(_fontList->GetSelection());
} else if (theChangedWidget == _ok) {
// apply font change and dismiss dialog
// ...
} else if (theChangedWidget == _cancel) {
// dismiss dialog
}
}
```

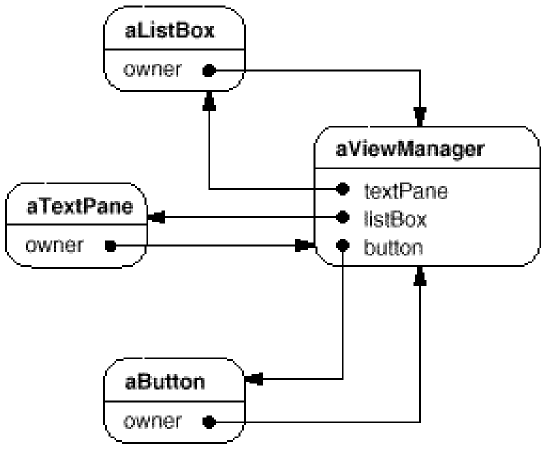
The complexity of WidgetChanged increases proportionallywith the complexity of the dialog. Large dialogs are undesirable forother reasons, of course, but mediator complexity might mitigate thepattern's benefits in other applications.

Known Uses

Both ET++ [WGM88] and the THINK C class library [Sym93b] usedirector-like objects in dialogs as mediators between widgets.

The application architecture of Smalltalk/V for Windows is based on amediator structure [LaL94]. In that environment, anapplication consists of a Window containing a set of panes. Thelibrary contains several predefined Pane objects; examples includeTextPane, ListBox, Button, and so on.These panes can be used without subclassing. An application developeronly subclasses from ViewManager, a class that's responsible for doinginter-pane coordination. ViewManager is the Mediator, and each paneonly knows its view manager, which is considered the "owner" of thepane. Panes don't refer to each other directly.

The following object diagram shows a snapshot of an application atrun-time:



Smalltalk/V uses an event mechanism for Pane-ViewManagercommunication. A pane generates an event when it wants to getinformation from the mediator or when it wants to inform the mediatorthat something significant happened. An event defines a symbol (e.g.,#select) that identifies the event. To handle the event, theview manager registers a method selector with the pane. This selectoris the event's handler; it will be invoked whenever the event occurs.

The following code excerpt shows how a ListPane object gets created insidea ViewManager subclass and how ViewManager registers an event handlerfor the #select event:

```
self addSubpane: (ListPane new
         paneName: 'myListPane';
         owner: self;
         when: #select perform: #listSelect:).
```

Another application of the Mediator pattern is in coordinating complexupdates. An example is the ChangeManager class mentioned in Observer (326). ChangeManager mediates betweensubjects and observers to avoid redundant updates. When an objectchanges, it notifies the ChangeManager, which in turn coordinates theupdate by notifying the object's dependents.

A similar application appears in the Unidraw drawingframework [VL90] and uses a class called CSolver toenforce connectivity constraints between "connectors." Objects ingraphical editors can appear to stick to one another in differentways. Connectors are useful in applications that maintainconnectivity automatically, like diagram editors and circuit designsystems. CSolver is a mediator between connectors. It solves theconnectivity constraints and updates the connectors' positions toreflect them.

Related Patterns

Facade (208) differsfrom Mediator in that it abstracts a subsystem of objects to providea more convenient interface. Its protocol is unidirectional; thatis, Facade objects make requests of the subsystem classes but notvice versa. In contrast, Mediator enables cooperative behaviorthat colleague objects don't or can't provide, and the protocol ismultidirectional.

Colleagues can communicate with the mediator using the Observer (326) pattern.