31

MEDIATOR

This pattern was previously described in GoF95.

DESCRIPTION

In general, object-oriented applications consist of a set of objects that interact with each other for the purpose of providing a service. This interaction can be direct (point-to-point) as long as the number of objects referring to each other directly is very low. Figure 31.1 depicts this type of direct interaction where `ObjectA` and `ObjectB` refer to each other directly.

As the number of objects increases, this type of direct interaction can lead to a complex maze of references among objects (Figure 31.2), which affects the maintainability of the application. Also, having an object directly referring to other objects greatly reduces the scope for reusing these objects because of higher coupling.

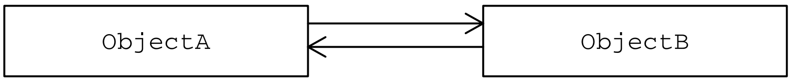


Figure 31.1 Point-to-Point Communication in the Case of Two Objects

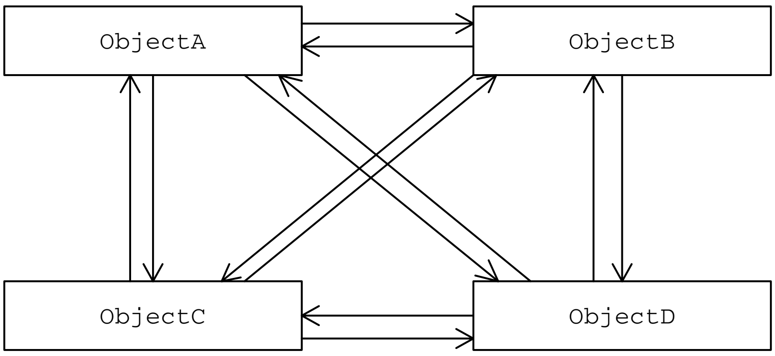


Figure 31.2 Point-to-Point Communication: Increased Number of Objects

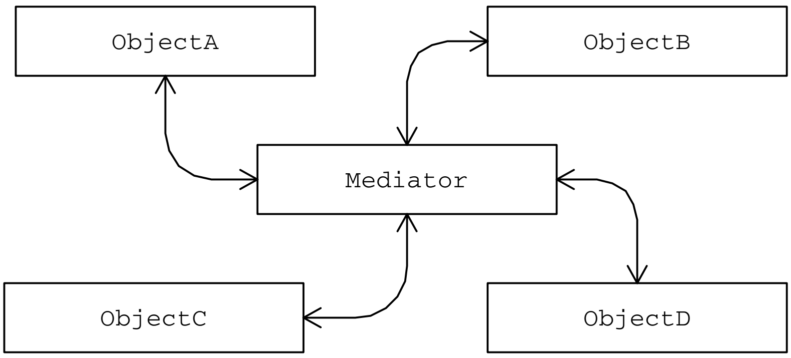


Figure 31.3 Object Interaction: `Mediator` as a Communication Hub

In such cases, the Mediator pattern can be used to design a controlled, coordinated communication model for a group of objects, eliminating the need for objects to refer to each other directly (Figure 31.3).

The Mediator pattern suggests abstracting all object interaction details into a separate class, referred to as a *Mediator*, with knowledge about the interacting group of objects. Every object in the group is still responsible for offering the service it is designed for, but objects do not interact with each other directly for this purpose. The interaction between any two different objects is routed through the `Mediator` class. All objects send their messages to the mediator. The mediator then sends messages to the appropriate objects as per the application's requirements. The resulting design has the following major advantages:

- With all the object interaction behavior moved into a separate (mediator) object, it becomes easier to alter the behavior of object interrelationships, by replacing the mediator with one of its subclasses with extended or altered functionality.
- Moving interobject dependencies out of individual objects results in enhanced object reusability.
- Because objects do not need to refer to each other directly, objects can be unit tested more easily.
- The resulting low degree of coupling allows individual classes to be modified without affecting other classes.

MEDIATOR VERSUS FAÇADE

In some aspects the Mediator pattern looks similar to the Façade pattern discussed earlier. Table 31.1 lists the similarities and differences between the two.

During the discussion of the Command pattern, we built two example applications. Let us revisit these applications and see how the direct object-to-object interaction can be avoided by applying the Mediator pattern.

Table 31.1 `Mediator` versus `Façade`

Mediator	*Façade*
A Mediator is used to abstract the necessary functionality of a group of objects with the aim of simplifying the object interaction.	A Façade is used to abstract the required functionality of a subsystem of components, with the aim of providing a simplified, higher level interface.
All objects interact with each other through the Mediator. The group of objects knows the existence of the Mediator.	Clients use the Façade to interact with subsystem components. The existence of the Façade is not known to the subsystem components.
Because the Mediator and all the objects that are registered with it can communicate with each other, the communication is bidirectional.	Clients can send messages (through the Façade) to the subsystem but not vice versa, making the communication unidirectional.
A Mediator can be assumed to stay in the middle of a group of interacting objects. Using a Mediator allows the implementation of any of the interacting objects to be changed without any impact on the other objects that interact with it only through the Mediator.	A Façade lies in between a client object and the subsystem. Using a Façade allows the implementation of the subsystem to be changed completely without any impact on its clients, provided the clients are not given direct access to the subsystem's classes.
By subclassing the Mediator, the behavior of the object interrelationships can be extended.	By subclassing the Façade, the implementation of the higher level interface can be changed.

EXAMPLE I

The FTP client simulation application built in the previous chapter has the following list of UI controls (Table 31.2) in the client display.

Figure 31.4 depicts the interaction between different UI objects.

Let us consider the following minor enhancements to the existing application to make it more user-friendly:

- When the UI is first displayed, all buttons except the `Exit` button should be disabled.
- When a file name is selected from the `JList` control displaying the local file system:
  - The `Upload` and `Delete` buttons should be enabled.
  - Any selected item in the remote file system display should be deselected.
  - The `Download` button should be disabled.
- When a file name is selected from the `JList` control displaying the remote file system:
  - The `Download` and `Delete` buttons should be enabled.
  - Any selected item in the local file system display should be deselected.
  - The `Upload` button should be disabled.

Table 31.2 List of User Interface Objects and the Associated Functionality

UI Control Object	*Functionality*
`JList`	Displays the local file system.
`JList`	Displays the remote file system.
`JButton`	Provides the upload functionality. When the `Upload` button is clicked, the selected file from the local file system is uploaded to the remote server and the file name is added to the remote file system `JList` control.
`JButton`	Provides the download functionality. When the `Download` button is clicked, the selected file from the remote file system is downloaded to the local system and the file name is added to the local file system display `JList` control.
`JButton`	Provides the delete functionality. When the `Delete` button is clicked, the selected file from the remote or local file system is deleted. The `JList` control is updated accordingly.

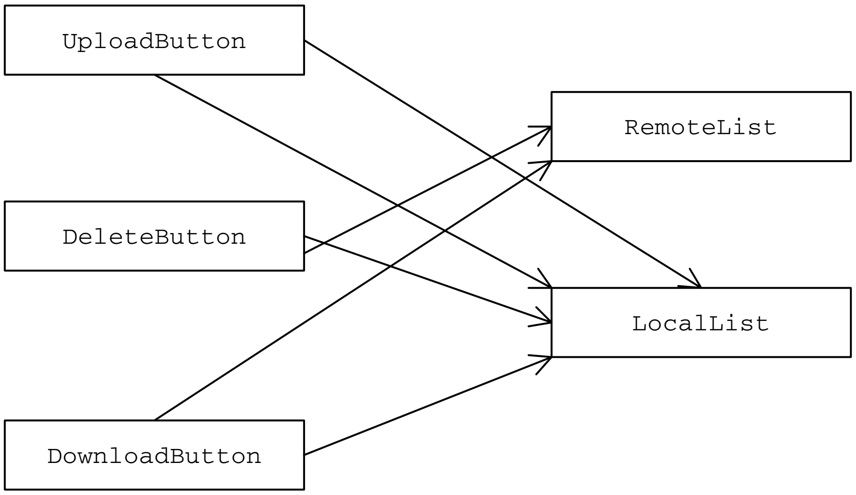


Figure 31.4 Object Interaction

– After executing the necessary upload/download operation, the `Upload`, `Download` and `Delete` buttons should be disabled. Similarly, after deleting the specified file, the `Delete` button should be disabled along with any `Upload` and `Download` buttons that are currently enabled. Both the local and remote file system displays should get refreshed after a delete, download or upload operation.

Figure 31.5 shows the resulting object interaction.

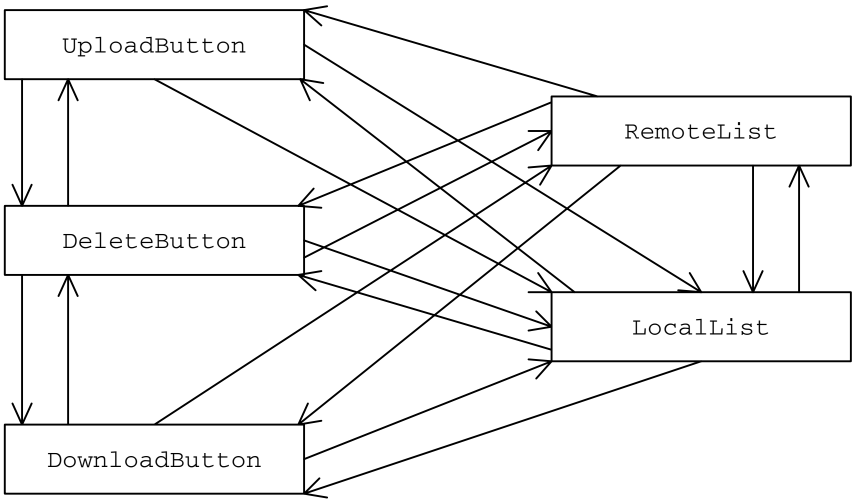


Figure 31.5 Object-to-Object Communication with Increased Direct Reference to Each Other

As more controls are added for additional functionality such as file rename, FTP server connect, disconnect and others, the direct communication between objects creates a complex maze of references among objects. This greatly reduces the maintainability of the application.

The Mediator pattern can be used in this case for a more efficient design of the object interaction. Applying the Mediator pattern, an abstraction for the object interaction details can be created. This abstraction can be designed as a separate `Mediator` class as in Figure 31.6 and Listing 31.1.

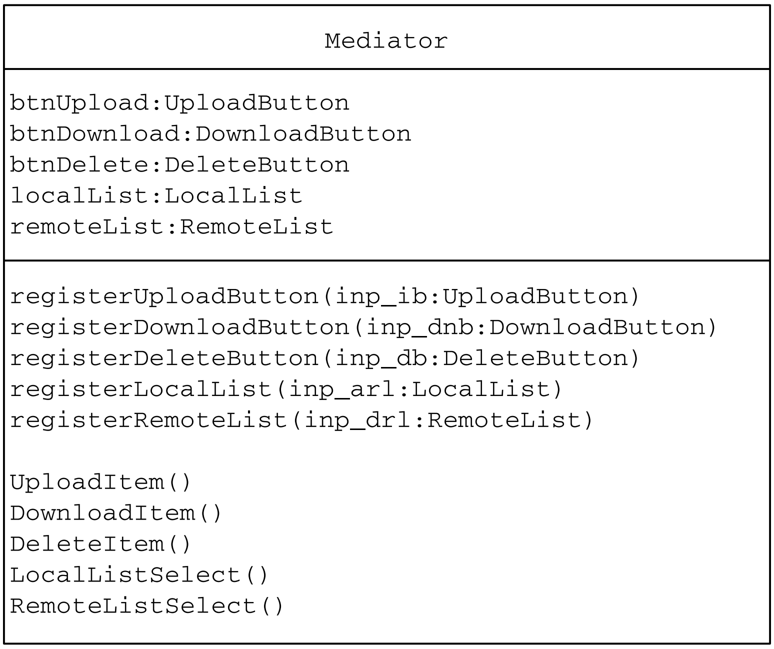
From the `Mediator` class implementation it can be seen that the `Mediator` offers methods for different UI objects to register themselves with the `Mediator`. The set of object interactions to be executed when each UI control is activated (or clicked) is designed as a separate method inside the `Mediator`.

Client Usage of the Mediator

The client (Listing 31.2) creates an instance of the `Mediator`. Whenever a `UI` object is created, the client passes the `Mediator` instance to it. The `UI` object registers itself with this instance of the `Mediator`.

User Interface Objects: Mediator Interaction

Because all the object interaction details are removed from individual `UI` objects to the `Mediator` object, the `processEvent` method of each of these `UI` objects gets reduced to a simple call to an appropriate `Mediator` method (Listing 31.3). Figure 31.7 shows the `UI` object interaction after the Mediator pattern is applied.



```
Mediator

btnUpload:UploadButton
btnDownload:DownloadButton
btnDelete:DeleteButton
localList:LocalList
remoteList:RemoteList

registerUploadButton(inp_ib:UploadButton)
registerDownloadButton(inp_dnb:DownloadButton)
registerDeleteButton(inp_db:DeleteButton)
registerLocalList(inp_arl:LocalList)
registerRemoteList(inp_drl:RemoteList)

UploadItem()
DownloadItem()
DeleteItem()
LocalListSelect()
RemoteListSelect()
```

Figure 31.6 **`Mediator`**

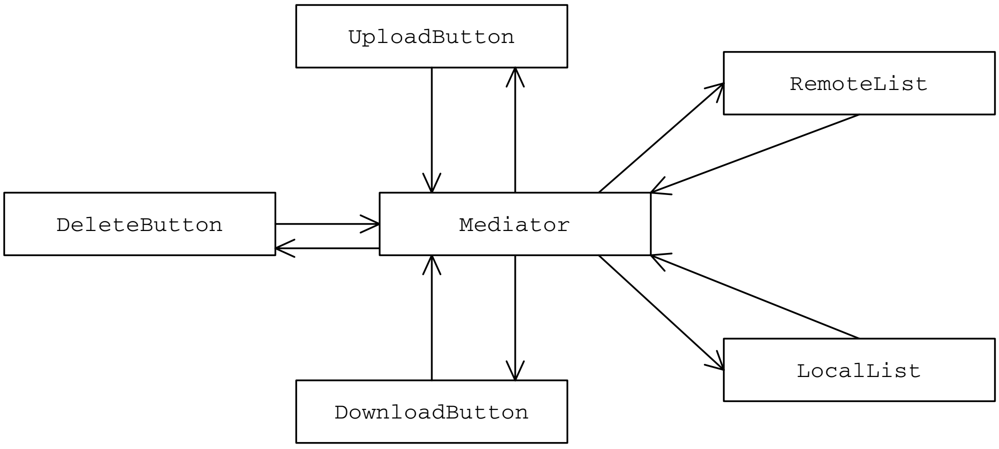


Figure 31.7 Object Interaction: `Mediator` as a Communication Hub

Listing 31.1 `Mediator` Class

```
class Mediator {
  private UploadButton btnUpload;
  private DownloadButton btnDownload;
  private DeleteButton btnDelete;
  private LocalList localList;
  private RemoteList remoteList;
  public void registerUploadButton(UploadButton inp_ib) {
    btnUpload = inp_ib;
  }
  public void registerDownloadButton(
    DownloadButton inp_dnb) {
    btnDownload = inp_dnb;
  }
  public void registerDeleteButton(DeleteButton inp_db) {
    btnDelete = inp_db;
  }
  public void registerLocalList(LocalList inp_arl) {
    localList = inp_arl;
  }
  public void registerRemoteList(RemoteList inp_drl) {
    remoteList = inp_drl;
  }
  public void UploadItem() {
    int index = localList.getSelectedIndex();
    String selectedItem =
      localList.getSelectedValue().toString();
    ((DefaultListModel) localList.getModel()).remove(
      index);
    ((DefaultListModel) remoteList.getModel()).addElement(
      selectedItem);
    btnUpload.setEnabled(false);
    btnDelete.setEnabled(false);
    btnDownload.setEnabled(false);
  }
```

(continued)

Listing 31.1 `Mediator` Class (Continued)

```
      public void DownloadItem() {
              …
              …
      }
      public void DeleteItem() {
              …
              …
      }
      public void LocalListSelect() {
              …
              …
      }
      public void RemoteListSelect() {
        localList.setSelectedIndex(-1);
        btnUpload.setEnabled(false);
        btnDelete.setEnabled(true);
        btnDownload.setEnabled(true);
      }
    }
```

EXAMPLE II

During the discussion of the Command pattern, we built an application to add and delete items to a library item database. A given item can be part of one or more categories. Each `Item` object maintains a list of all categories which it is part of. Similarly, each `Category` object maintains a list of all items that currently are part of it. The class association diagram in Figure 31.8 depicts this relationship.

When an application has to deal with many items that belong to one or more categories, the object interactions can get complicated. The diagram in Figure 31.9 depicts a scenario where different `Item` and `Category` objects refer to each other directly.

The direct interaction between different `Item` objects and `Category` objects can be eliminated by moving the object interaction details out of the `Item` and `Category` classes to a separate `Mediator` class (Figure 31.10). The `Mediator` can be designed with the following two sets of methods:

- A set of methods to allow different `Item` and `Category` objects to register with the `Mediator`.
- A set of methods for adding and deleting items. The `Mediator` is responsible for implementing interactions between different objects as part of these methods.

Listing 31.2 Client `FTPGUI` Class

```
public class FTPGUI extends JFrame {
          …
          …
  private Mediator mdtr = new Mediator();
  public FTPGUI() throws Exception {
          …
          …
    //Create controls
    defLocalList = new DefaultListModel();
    defRemoteList = new DefaultListModel();
    localList = new LocalList(defLocalList, mdtr);
    remoteList = new RemoteList(defRemoteList, mdtr);
    pnlFTPUI = new JPanel();
          …
          …
    //Create buttons
    UploadButton btnUpload =
      new UploadButton(FTPGUI.UPLOAD, mdtr);
    btnUpload.setMnemonic(KeyEvent.VK_U);
    DownloadButton btnDownload =
      new DownloadButton(FTPGUI.DOWNLOAD, mdtr);
    btnDownload.setMnemonic(KeyEvent.VK_N);
    DeleteButton btnDelete =
      new DeleteButton(FTPGUI.DELETE, mdtr);
    btnDelete.setMnemonic(KeyEvent.VK_D);
          …
          …
  }
          …
          …
}//end of class
```

The `Mediator` can maintain the `Item-Category` association in the `itemCatAssoc` instance variable. `Item` objects do not need to refer to `Category` objects directly. Hence an `Item` object does not need to maintain the list of `Categories` it belongs to and vice versa. Similarly, both add and delete operations are not required to be implemented by the `Item` and the `Category` classes.

The `execute` method of the `AddCommand` and `DeleteCommand` `Command` objects gets reduced to a call to the `addItem` and `deleteItem` `Mediator` methods, respectively.

Listing 31.3 Simplified `UI` Object Classes

```
          …
          …
class UploadButton extends JButton
  implements CommandInterface {
  Mediator mdtr;
  public void processEvent() {
    mdtr.UploadItem();
  }
  public UploadButton(String name, Mediator inp_mdtr) {
    super(name);
    mdtr = inp_mdtr;
    mdtr.registerUploadButton(this);
  }
}
class DownloadButton extends JButton
  implements CommandInterface {
  Mediator mdtr;
  public void processEvent() {
    mdtr.DownloadItem();
  }
  public DownloadButton(String name, Mediator inp_mdtr) {
    super(name);
    mdtr = inp_mdtr;
    mdtr.registerDownloadButton(this);
  }
}
          …
          …
```

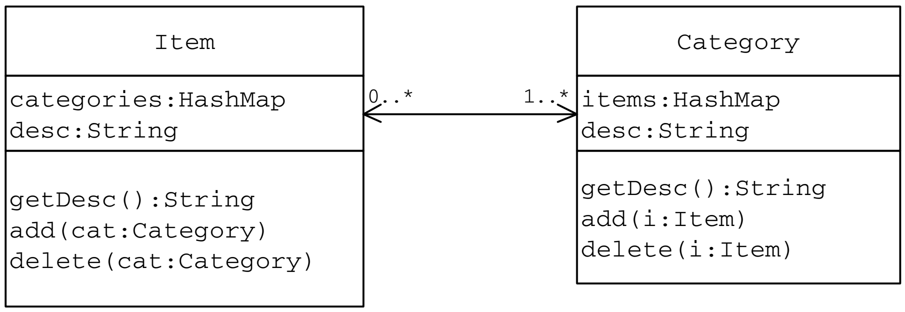


Figure 31.8 **`Item-Category`** **Association**

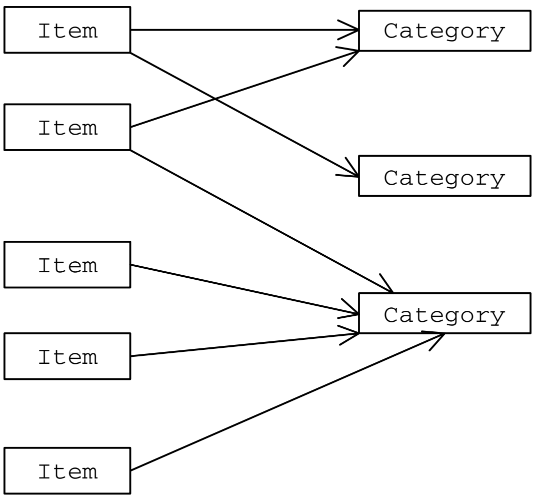


Figure 31.9 **`Item-Category`** **Object Interaction**

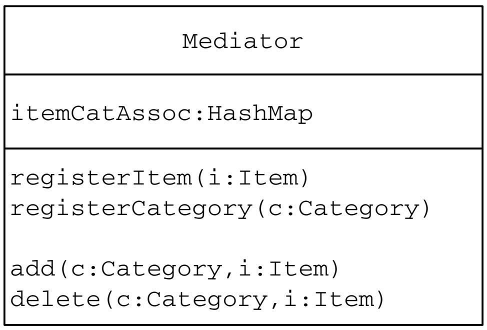


Figure 31.10 **`Mediator`**

PRACTICE QUESTIONS

1. Customer service representatives at some commercial banks handle queries from their existing and potential customers using an online chat application. At peak times, each representative may need to work with more than one customer simultaneously. Design this communication mechanism with a `Mediator` object between different `User` objects and `Representative` objects.
2. Implement the Example II application.