30

COMMAND

This pattern was previously described in GoF95.

DESCRIPTION

In general, an object-oriented application consists of a set of interacting objects each offering limited, focused functionality. In response to user interaction, the application carries out some kind of processing. For this purpose, the application makes use of the services of different objects for the processing requirement. In terms of implementation, the application may depend on a designated object that invokes methods on these objects by passing the required data as arguments (Figure 30.1). This designated object can be referred to as an *invoker* as it invokes operations on different objects. The invoker may be treated as part of the client application. The set of objects that actually contain the implementation to offer the services required for the request processing can be referred to as *Receiver* objects.

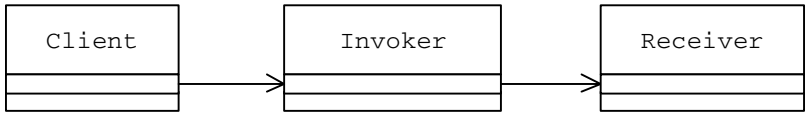

Figure 30.1 Object Interaction: Before Applying the Command Pattern

In this design, the application that forwards the request and the set of `Receiver` objects that offer the services required to process the request are closely tied to each other in that they interact with each other directly. This could result in a set of conditional *if* statements in the implementation of the invoker.

```
    …
if (RequestType=TypeA){
  //do something
}
```

```
if (RequestType=TypeB){
  //do something
}
     …
```

When a new type of feature is to be added to the application, the existing code needs to be modified and it violates the basic object-oriented open-closed principle.

```
     …
if (RequestType=TypeA){
  //do something
}
     …
if (RequestType=NewType){
  //do something
}
     …
```

The open-closed principle states that a software module should be:

- *Open for extension* — It should be possible to alter the behavior of a module or add new features to the module functionality.
- *Closed for modification* — Such a module should not allow its code to be modified.

In a nutshell, the open-closed principle helps in designing software modules whose functionality can be extended without having to modify the existing code.

Using the Command pattern, the invoker that issues a request on behalf of the client and the set of service-rendering `Receiver` objects can be decoupled. The Command pattern suggests creating an abstraction for the processing to be carried out or the action to be taken in response to client requests.

This abstraction can be designed to declare a common interface to be implemented by different concrete implementers referred to as *Command objects*. Each `Command` object represents a different type of client request and the corresponding processing. In Figure 30.2, the `Command` interface represents the abstraction. It declares an `execute` method, which is implemented by two of its implementer (command) classes — `ConcreteCommand_1` and `ConcreteCommand_2`.

A given `Command` object is responsible for offering the functionality required to process the request it represents, but it does not contain the actual implementation of the functionality. `Command` objects make use of `Receiver` objects in offering this functionality (Figure 30.3).

When the client application needs to offer a service in response to user (or other application) interaction:

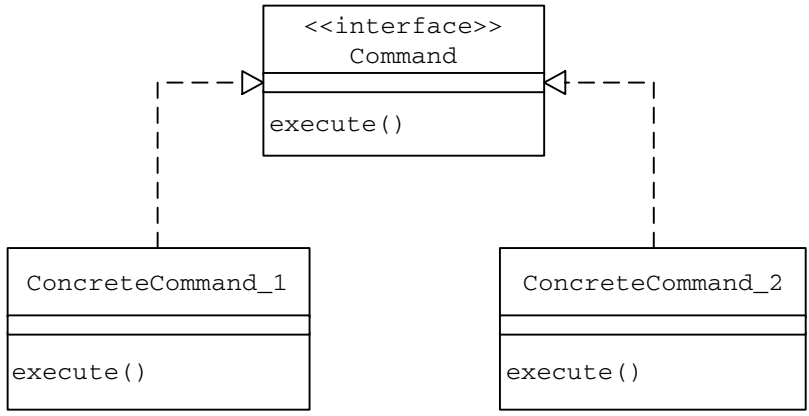

Figure 30.2 `Command` Object Hierarchy

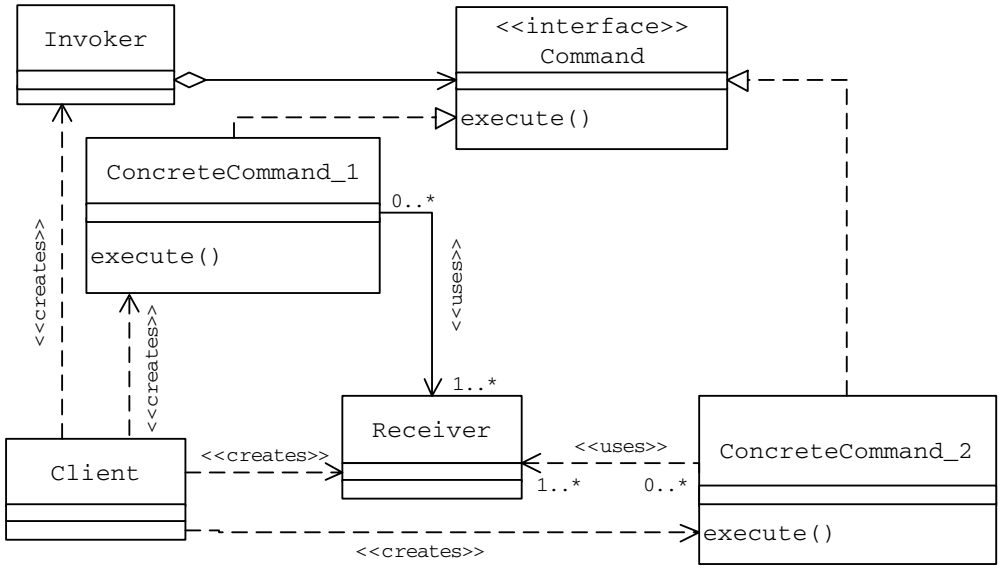

Figure 30.3 Class Association: After the Command Pattern Is Applied

1. It creates the necessary `Receiver` objects.
2. It creates an appropriate `Command` object and configures it with the `Receiver` objects created in Step 1.
3. It creates an instance of the invoker and configures it with the `Command` object created in Step 2.
4. The invoker invokes the `execute()` method on the Command object.
5. As part of its implementation of the `execute` method, a typical `Command` object invokes necessary methods on the `Receiver` objects it contains to provide the required service to its caller.

In the new design:

- The client/invoker does not directly interact with `Receiver` objects and therefore, they are completely decoupled from each other.
- When the application needs to offer a new feature, a new `Command` object can be added. This does not require any changes to the code of the invoker. Hence the new design conforms to the open-closed principle.
- Because the request is designed in the form of an object, it opens up a whole new set of possibilities such as:
  - Storing a `Command` object to persistent media:
    - To be executed later.
    - To apply reverse processing to support the undo feature.
  - Grouping together different `Command` objects to be executed as a single unit.

The following FTP (File Transfer Protocol) client example application provides a good understanding of how the `Command` pattern can be applied in real world applications.

EXAMPLE I

Let us build an application that simulates the working of an FTP client. In Java, a simple FTP client user interface can be designed using:

- Two `JList` objects for the local and remote file systems display
- Four `JButton` objects for initiating different types of requests such as upload, download, delete and exit

Once the user interface controls are arranged in a frame, the UI display looks as in Figure 30.4.

When each of the `JButton` objects is created, an instance of the `ButtonHandler` class that implements the built-in `ActionListener` interface is set as its `ActionListener`. This means that whenever a `JButton` object in the UI display is clicked, the `actionPerformed` method of the `ButtonHandler` object gets executed.

```
public class FTPGUI extends JFrame {
    …
    …
  //Create buttons
  btnUpload = new JButton(FTPGUI.UPLOAD);
  btnUpload.setMnemonic(KeyEvent.VK_U);
    …
    …
  ButtonHandler vf = new ButtonHandler();
  btnUpload.addActionListener(vf);
  btnDownload.addActionListener(vf);
```

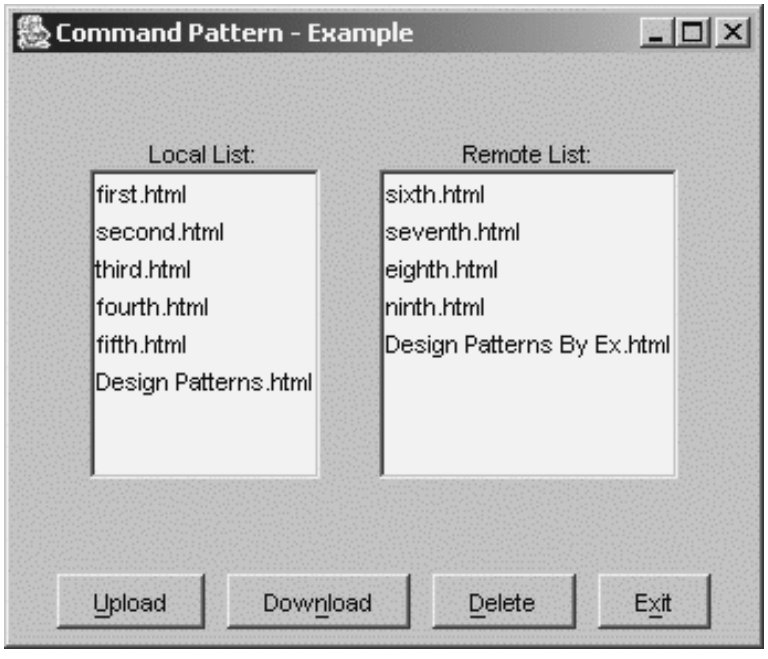

Figure 30.4 Simple FTP Client UI Display

```
    btnDelete.addActionListener(vf);
    btnExit.addActionListener(vf);
        …
        …
}//end of class
```

Because the same instance of the `ButtonHandler` is set as the `ActionListener` for all `JButton` objects in the UI display, the `actionPerformed` method is called for all `JButton` objects. Hence the `ButtonHandler` object must check which button is clicked and carry out the appropriate processing.

From Listing 30.1, it can be seen that code in the `actionPerformed` method is a little inelegant with a set of conditional statements and as more button and menu item objects are added to the FTP UI, the code could quickly become cluttered. Also, when a new button object is to be added, the existing code in the `actionPerformed` method needs to be modified. This violates the object-oriented open-closed principle.

Let us redesign the application using the Command pattern. Applying the Command pattern, let us define an abstraction in the form of a `CommandInterface` interface for the functionality associated with different button objects in the FTP client UI.

```
interface CommandInterface {
  public void processEvent();
}
```

Listing 30.1 `ButtonHandler` Class

```
class ButtonHandler implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    //if statements - for different types of client requests
    if (e.getActionCommand().equals(FTPGUI.EXIT)) {
      System.exit(1);
    }
    if (e.getActionCommand().equals(FTPGUI.UPLOAD)) {
      int index = localList.getSelectedIndex();
      String selectedItem =
        localList.getSelectedValue().toString();
      ((DefaultListModel) localList.getModel()).remove(
        index);
      ((DefaultListModel) remoteList.getModel()).
      addElement(selectedItem);
    }
    if (e.getActionCommand().equals(FTPGUI.DOWNLOAD)) {
      int index = remoteList.getSelectedIndex();
      String selectedItem =
        remoteList.getSelectedValue().toString();
      ((DefaultListModel) remoteList.getModel()).remove(
        index);
      ((DefaultListModel) localList.getModel()).
      addElement(selectedItem);
    }
    if (e.getActionCommand().equals(FTPGUI.DELETE)) {
      int index = localList.getSelectedIndex();
      if (index >= 0) {
        ((DefaultListModel) localList.getModel()).
        remove(index);
      }
     index = remoteList.getSelectedIndex();
      if (index >= 0) {
        ((DefaultListModel) remoteList.getModel()).
        remove(index);
      }
    }
  }
}
```

Different button objects themselves can implement this interface and behave as individual command objects. But this is not recommended as:

- The `JButton` class is a highly reusable class and is used on many occasions where the Java Swing library is used to create an application user interface. Implementation specific to the `CommandInterface` may not be applicable in all such cases.
- If the `JButton` class is redesigned to implement the `CommandInterface` interface, it needs to implement the functionality required to process different types of requests such as upload, download and others corresponding to different `JButton` objects in the user interface. This results in adding unrelated functionality to the `JButton` class — low cohesion. In addition:
  - This could lead to inelegant conditional statements.
  - Every time a new button is added to the user interface, it would require changes to the existing implementation of the `processEvent` method, which is a violation of the object-oriented open-closed principle.

To overcome these problems, a set of new button classes, each corresponding to a different type of request, can be designed as subclasses of the `JButton` class (Listing 30.2). These subclasses can be designed to implement the `CommandInterface`. As part of its implementation of the `processEvent` method, each subclass of the `JButton` class offers the functionality required to process the request it represents (Figure 30.5).

The FTP UI can be built using objects of this new set of `JButton` subclasses. The rest of the application remains unchanged and the `actionPerformed` method gets highly simplified to a mere two lines of code.

```
class buttonHandler implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    CommandInterface CommandObj =
      (CommandInterface) e.getSource();
    CommandObj.processEvent();
  }
}
```

In the new design whenever a new button or a menu item is to be added, a new `Command` object needs to be created as an implementer of the `CommandInterface`. The new `Command` object can be added to the application in a seamless manner without requiring changes to the existing `actionPerformed` method code. On the negative side, this results in a larger number of classes.

Listing 30.2 `JButton` Subclasses to Perform Different FTP Operations

```
class UploadButton extends JButton
  implements CommandInterface {
  public void processEvent() {
    int index = localList.getSelectedIndex();
    String selectedItem =
      localList.getSelectedValue().toString();
    ((DefaultListModel) localList.getModel()).remove(
      index);
    ((DefaultListModel) remoteList.getModel()).addElement(
      selectedItem);
  }
  public UploadButton(String name) {
    super(name);
  }
}
class DownloadButton extends JButton
  implements CommandInterface {
  public void processEvent() {
    int index = remoteList.getSelectedIndex();
    String selectedItem =
      remoteList.getSelectedValue().toString();
    ((DefaultListModel) remoteList.getModel()).remove(
      index);
    ((DefaultListModel) localList.getModel()).addElement(
      selectedItem);
  }
  public DownloadButton(String name) {
    super(name);
  }
}
class DeleteButton extends JButton
  implements CommandInterface {
  public void processEvent() {
    int index = localList.getSelectedIndex();
    if (index >= 0) {
      ((DefaultListModel) localList.getModel()).remove(
        index);
    }
```

(continued)

Listing 30.2 `JButton` Subclasses to Perform Different FTP Operations (Continued)

```
        index = remoteList.getSelectedIndex();
        if (index >= 0) {
          ((DefaultListModel) remoteList.getModel()).remove(
            index);
        }
      }
      public DeleteButton(String name) {
          super(name);
      }
    }
    class ExitButton extends JButton
      implements CommandInterface {
      public void processEvent() {
        System.exit(1);
      }
      public ExitButton(String name) {
        super(name);
      }
    }
```

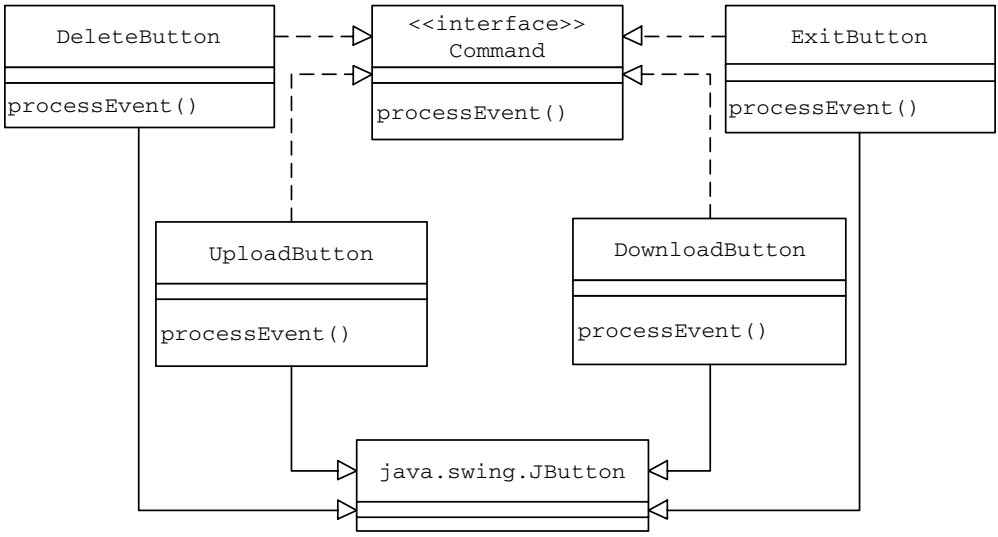

Figure 30.5 FTP UI: `Command` Object Hierarchy

EXAMPLE II

Let us build an application to manage items in a library item database. Typical library items include books, CDs, videos and DVDs. These items are grouped into categories and a given item can belong to one or more categories. For example, a new movie video may belong to both the `Video` category and the `NewReleases` category.

Let us define two classes — `Item` and `Category` — (Listing 30.3) representing a typical library item and a category of items, respectively (Figure 30.6).

From the design and the implementation of the `Item` and the `Category` classes, it can be seen that a `Category` object maintains a list of its current member items. Similarly, an `Item` object maintains the list of all categories which it is part of. For simplicity, let us suppose that the library item management application deals only with adding and deleting items. Applying the Command pattern, the action to be taken to process *add item* and *delete item* requests can be designed as implementers of a common `CommandInterface` interface. The `CommandInterface` provides an abstraction for the processing to be carried out in response to a typical library item management request such as add or delete item. The `CommandInterface` implementers — `AddCommand` and `DeleteCommand` — in Figure 30.7 represent the add and the delete item request, respectively.

Let us further define an invoker `ItemManager` class.

```
public class ItemManager {
  CommandInterface command;
  public void setCommand(CommandInterface c) {
    command = c;
  }
  public void process() {
    command.execute();
  }
}
```

The `ItemManager`:

- Contains a `Command` object within
- Invokes the `Command` object's `execute` method as part of its `process` method implementation
- Provides a `setCommand` method to allow client objects to configure it with a `Command` object

The client `CommandTest` uses the invoker `ItemManager` to get its *add item* and *delete item* requests processed.

Application Flow

To add or delete an item, the client `CommandTest` (Listing 30.4):

1. Creates the necessary `Item` and `Category` objects. These objects act as receivers.
2. Creates an appropriate `Command` object that corresponds to its current request. The set of `Receiver` objects created in Step 1 is passed to the `Command` object at the time of its creation.
3. Creates an instance of the `ItemManager` and configures it with the `Command` object created in Step 2.
4. Invokes the `process()` method of the `ItemManager`. The `ItemManager` invokes the `execute` method on the `Command` object. The `Command` object in turn invokes necessary `Receiver` object methods. Different `Item` and `Category` `Receiver` objects perform the actual request processing. To keep the example simple, no database access logic is implemented. Both `Item` and `Category` objects are implemented to simply display a message.

When the client program is run, the following output is displayed:

```
Item 'A Beautiful Mind' has been added to the 'CD' Category
Item 'Duet' has been added to the 'CD' Category
Item 'Duet' has been added to the 'New Releases' Category
Item 'Duet' has been deleted from the 'New Releases'
Category
```

The class diagram in Figure 30.8 depicts the overall class association.

The sequence diagram in Figure 30.9 shows the message flow when the client `CommandTest` uses a `Command` object to add an item.

PRACTICE QUESTIONS

1. In Example I above, different concrete `Command` classes are designed as inner classes. Redesign and implement the example application with different `Command` classes as external classes.
2. Add a new method `undo()` to the `CommandInterface` in Examples I and II. Enhance different command classes implementing this method to offer the functionality required to undo the effect of the `execute()` method.
3. Enhance the Example II application to include the ability to log the data and time when a specific add or delete operation is performed.
4. Enhance the Example II application to add the *move* functionality that allows an item to be moved from one category to another. Implement the *move* functionality as a combination of *delete* followed by an *add* operation. Both delete and add operations must be executed together as a unit to provide the move functionality.

Listing 30.3 Item and Category Classes

```
public class Item {
  private HashMap categories;
  private String desc;
  public Item(String s) {
    desc = s;
    categories = new HashMap();
  }
  public String getDesc() {
    return desc;
  }
  public void add(Category cat) {
    categories.put(cat.getDesc(), cat);
  }
  public void delete(Category cat) {
    categories.remove(cat.getDesc());
  }
}
public class Category {
  private HashMap items;
  private String desc;
  public Category(String s) {
    desc = s;
    items = new HashMap();
  }
  public String getDesc() {
    return desc;
  }
  public void add(Item i) {
    items.put(i.getDesc(), i);
    System.out.println("Item '" + i.getDesc() +
                       "' has been added to the '" +
                       getDesc() + "' Category ");
  }
```

(continued)

Listing 30.3 Item and Category Classes (Continued)

```
    public void delete(Item i) {
      items.remove(i.getDesc());
      System.out.println("Item '" + i.getDesc() +
                         "' has been deleted from the '" +
                         getDesc() + "' Category ");
    }
  }
```

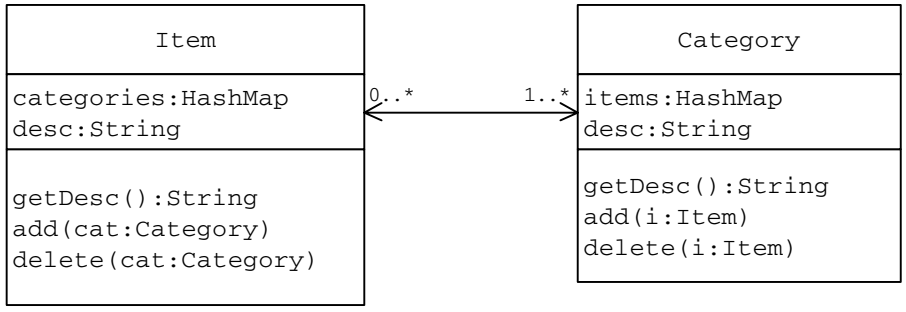

Figure 30.6 Item-Category Association

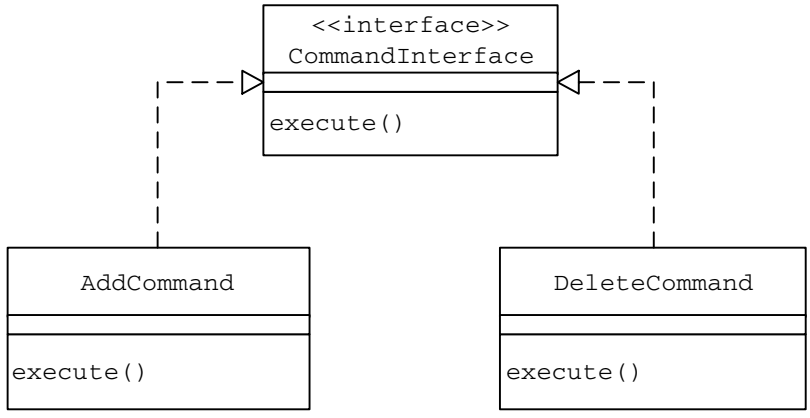

Figure 30.7 `Command` Object Hierarchy

Listing 30.4 Client `CommandTest` Class

```
public class CommandTest {
  public static void main(String[] args) {
    //Add an item to the CD category
    //create Receiver objects
    Item CD = new Item("A Beautiful Mind");
    Category catCD = new Category("CD");
    //create the command object
    CommandInterface command = new AddCommand(CD, catCD);
    //create the invoker
    ItemManager manager = new ItemManager();
    //configure the invoker
    //with the command object
    manager.setCommand(command);
    manager.process();
    //Add an item to the CD category
    CD = new Item("Duet");
    catCD = new Category("CD");
    command = new AddCommand(CD, catCD);
    manager.setCommand(command);
    manager.process();
    //Add an item to the New Releases category
    CD = new Item("Duet");
    catCD = new Category("New Releases");
   command = new AddCommand(CD, catCD);
   manager.setCommand(command);
   manager.process();
    //Delete an item from the New Releases category
   command = new DeleteCommand(CD, catCD);
    manager.setCommand(command);
    manager.process();
  }
}
```

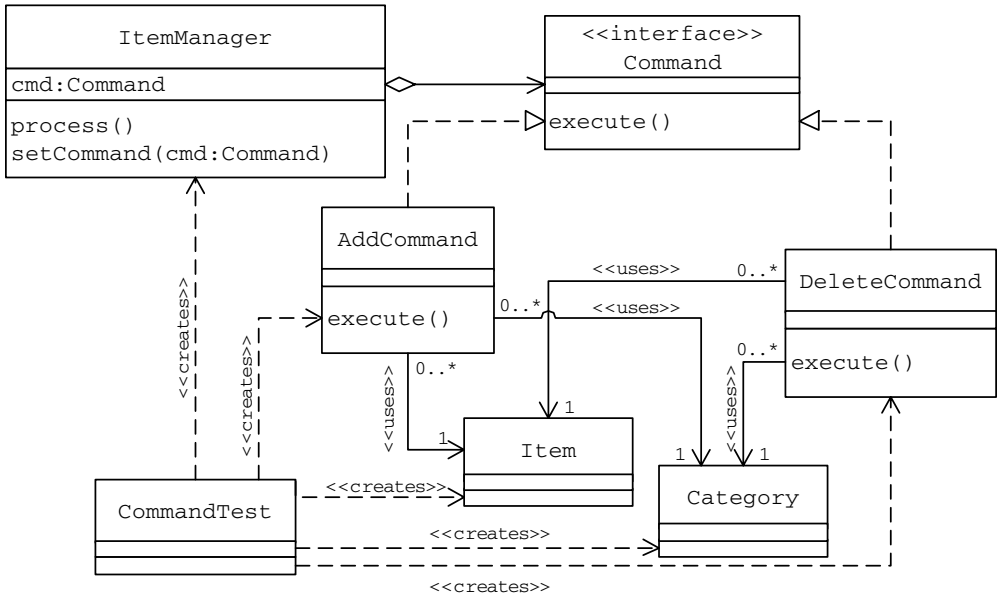

Figure 30.8 Class Association

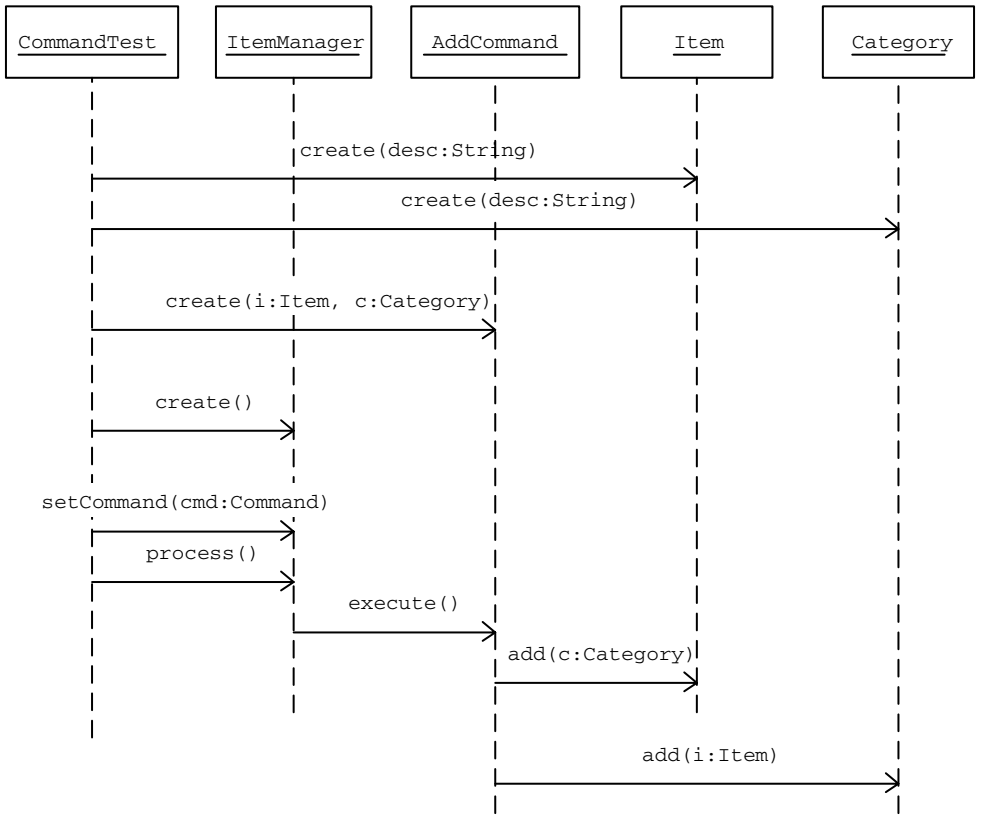

Figure 30.9 Message Flow When an Item Is Added to a Category