33

OBSERVER

This pattern was previously described in GoF95.

DESCRIPTION

The Observer pattern is useful for designing a consistent communication model between a set of dependent objects and an object that they are dependent on. This allows the dependent objects to have their state synchronized with the object that they are dependent on. The set of dependent objects are referred to as *observers* and the object that they are dependent on is referred to as the *subject*. In order to accomplish this, the Observer pattern suggests a *publisher-subscriber* model leading to a clear boundary between the set of `Observer` objects and the `Subject` object.

A typical observer is an object with interest or dependency in the state of the subject. A subject can have more than one such observer. Each of these observers needs to know when the subject undergoes a change in its state.

The subject cannot maintain a static list of such observers as the list of observers for a given subject could change dynamically. Hence any object with interest in the state of the subject needs to explicitly register itself as an observer with the subject. Whenever the subject undergoes a change in its state, it notifies all of its registered observers. Upon receiving notification from the subject, each of the observers queries the subject to synchronize its state with that of the subject's. Thus a subject behaves as a publisher by publishing messages to all of its subscribing observers.

In other words, the scenario contains a one-to-many relationship between a subject and the set of its observers. Whenever the subject instance undergoes a state change, all of its dependent observers are notified and they can update themselves. Each of the observer objects has to register itself with the subject to get notified when there is a change in the subject's state. An observer can register or subscribe with multiple subjects. Whenever an observer does not wish to be notified any further, it unregisters itself with the subject.

For this mechanism to work:

- The subject should provide an interface for registering and unregistering for change notifications.

- One of the following two must be true:
  - *In the pull model* — The subject should provide an interface that enables observers to query the subject for the required state information to update their state.
  - *In the push model* — The subject should send the state information that the observers may be interested in.
- Observers should provide an interface for receiving notifications from the subject.

The class diagram in Figure 33.1 describes the structure of different classes and their association, catering to the above list of requirements.

From this class diagram it can be seen that:

- All subjects are expected to provide implementation for an interface similar to the `Observable` interface.
- All observers are expected to have an interface similar to the `Observer` interface.

Several variations can be thought of while applying the Observer pattern, leading to different types of subject-observers such as observers that are interested only in specific types of changes in the subject.

ADDING NEW OBSERVERS

After applying the Observer pattern, different observers can be added dynamically without requiring any changes to the `Subject` class. Similarly, observers remain unaffected when the state change logic of the subject changes.

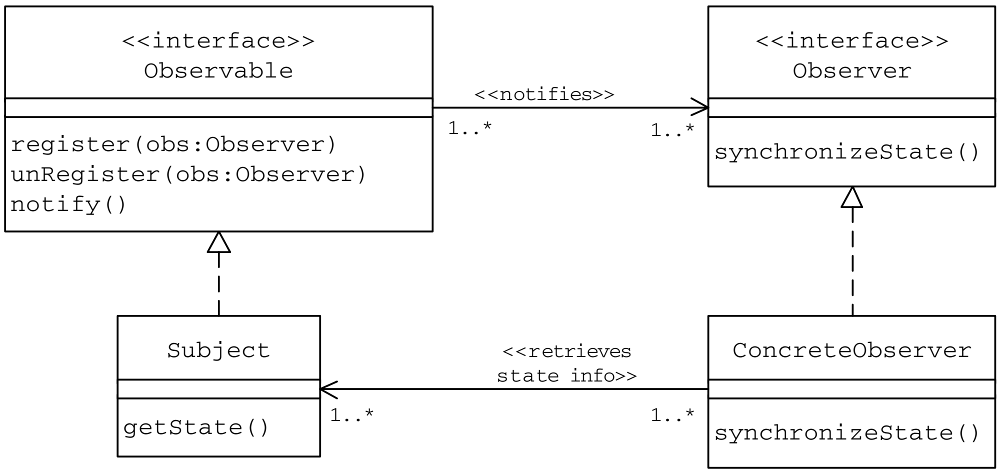


Figure 33.1 Generic Class Association When the Observer Pattern Is Applied

EXAMPLE

Let us build a sales reporting application for the management of a store with multiple departments. The features of the application include:

- Users should be able to select a specific department they are interested in.
- Upon selecting a department, two types of reports are to be displayed:
  - *Monthly report* — A list of all transactions for the current month for the selected department.
  - *YTD sales chart* — A chart showing the year-to-date sales for the selected department by month.
- Whenever a different department is selected, both of the reports should be refreshed with the data for the currently selected department.

From the proposed functionality described above, we can easily see that two of the reporting objects are dependent upon the object that carries the user-selected department. We can apply the Observer pattern in this case to design a consistent communication model between the object holding the user selection and both of the dependent report objects.

Let us define three classes with the stated functionality as in Table 33.1.

Applying the Observer pattern, let us define an Observable interface to be implemented by the ReportManager (Figure 33.2).

```
public interface Observable {
  public void notifyObservers();
  public void register(Observer obs);
  public void unRegister(Observer obs);
}
```

Table 33.1 Subject-Observer Classes

Class	*Role*	*Functionality*
ReportManager	Subject	Displays the necessary UI for the user to select a department. Maintains the user selected department in an instance variable.
MonthlyReport	Observer	Displays the monthly report for the selected department.
YTDChart	Observer	Displays the YTD sales chart for the selected department.

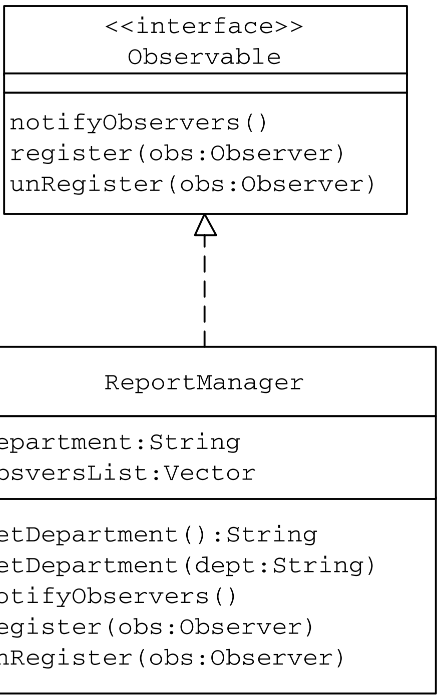


Figure 33.2 Observable Interface and Its Implementer

The ReportManager class (Listing 33.1) provides implementation for methods declared in the Observable interface. Both of the dependent report objects can use these methods to register themselves as observers. The ReportManager stores each of these registered observers in the observersList vector. The currently selected department constitutes the state of the ReportManager object and is maintained in the form of an instance variable named department. Whenever a new value is set for the department variable (this constitutes a change in the state), the notifyObservers method is invoked. As part of the notifyObservers method, the ReportManager invokes the refreshData(Observable) method on each of its currently registered observers.

Besides providing an implementation for the Observable interface methods, the ReportManager displays the necessary user interface as in Figure 33.3 to allow a user to select a specific department of interest.

Let us also define an interface Observer to be implemented by both the MonthlyReport and the YTDChart classes (Figure 33.4 and Listing 33.2):

```
public interface Observer {
  public void refreshData(Observable subject);
}
```

The ReportManager makes use of this interface to notify its observers.

Listing 33.1 `ReportManager` Class

```
public class ReportManager extends JFrame
  implements Observable {
          …
          …
  private Vector observersList;
  private String department;
  public ReportManager() throws Exception {
          …
          …
    observersList = new Vector();
          …
          …
  }
  public void register(Observer obs) {
    //Add to the list of Observers
    observersList.addElement(obs);
  }
  public void unRegister(Observer obs) {
    //remove from the list of Observers
  }
  public void notifyObservers() {
    //Send notify to all Observers
    for (int i = 0; i < observersList.size(); i++) {
      Observer observer =
        (Observer) observersList.elementAt(i);
      observer.refreshData(this);
    }
  }
  public String getDepartment() {
    return department;
  }
  public void setDepartment(String dept) {
    department = dept;
  }
```

(continued)

Listing 33.1 `ReportManager` Class (Continued)

```
    class ButtonHandler implements ActionListener {
      ReportManager subject;
      public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals(ReportManager.EXIT)) {
          System.exit(1);
        }
        if (e.getActionCommand().equals(ReportManager.SET_OK)) {
          String dept = (String)
                          cmbDepartmentList.getSelectedItem();
          //change in state
          subject.setDepartment(dept);
          subject.notifyObservers();
        }
      }
      public ButtonHandler() {
      }
      public ButtonHandler(ReportManager manager) {
        subject = manager;
      }
    }
  }//end of class
```

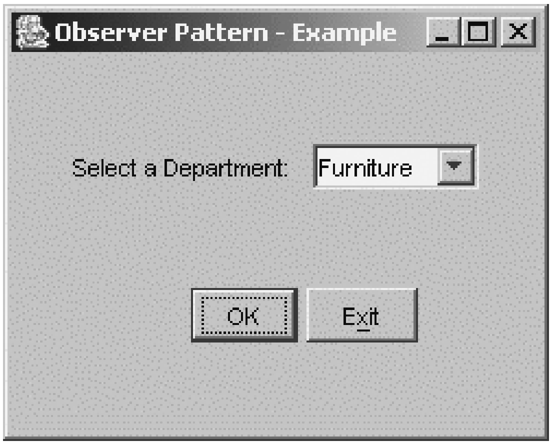


Figure 33.3 `ReportManager` User Interface

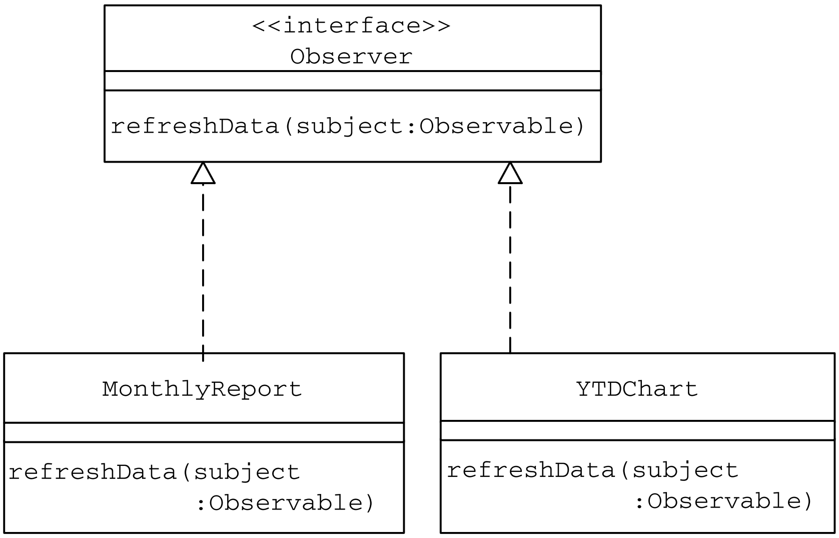


Figure 33.4 Observer Class Hierarchy

Subject–Observer Association

Typically, a client would first create an instance of the subject (ReportManager). Whenever an Observer (e.g., MonthlyReport, YTDChart) object is created, it passes the Subject instance reference to it as part of its Constructor method call. The Observer object registers itself with this Subject instance.

```
//Client Code
public class SupervisorView {
          …
          …
  public static void main(String[] args) throws Exception {
    //Create the Subject
    ReportManager objSubject = new ReportManager();
    //Create Observers
    new MonthlyReport(objSubject);
    new YTDChart(objSubject);
  }
}//end of class
```

The resulting class association can be depicted as in Figure 33.5.

Listing 33.2 `MonthlyReport` Class as an Observer

```
public class MonthlyReport extends JFrame implements Observer {
          …
          …
  private ReportManager objReportManager;
  public MonthlyReport(ReportManager inp_objReportManager)
  throws Exception {
    super("Observer Pattern - Example");
    objReportManager = inp_objReportManager;
    //Create controls
          …
          …
    //Create Labels
          …
          …
    objReportManager.register(this);
  }
  public void refreshData(Observable subject) {
    if (subject == objReportManager) {
      //get subject's state
      String department = objReportManager.getDepartment();
      lblTransactions.setText(
        "Current Month Transactions - " +
        department);
      Vector trnList =
        getCurrentMonthTransactions(department);
      String content = "";
      for (int i = 0; i < trnList.size(); i++) {
        content = content +
                  trnList.elementAt(i).toString() + "\n";
      }
      taTransactions.setText(content);
    }
  }
  private Vector getCurrentMonthTransactions(String department
                                                        ) {
```

(continued)

Listing 33.2 `MonthlyReport` Class as an Observer (Continued)

```
    Vector v = new Vector();
    FileUtil futil = new FileUtil();
    Vector allRows = futil.fileToVector("Transactions.date");
    //current month
    Calendar cal = Calendar.getInstance();
    cal.setTime(new Date());
    int month = cal.get(Calendar.MONTH) + 1;
    String searchStr = department + ″," + month + ″,";
    int j = 1;
    for (int i = 0; i < allRows.size(); i++) {
      String str = (String) allRows.elementAt(i);
      if (str.indexOf(searchStr) > -1) {
        StringTokenizer st =
          new StringTokenizer(str, ″,");
        st.nextToken();//bypass the department
        str = " " + j + ″. " + st.nextToken() + "/" +
                st.nextToken() + "~~~" +
                st.nextToken() + "Items" + "~~~" +
                st.nextToken() + " Dollars";
        j++;
        v.addElement(str);
      }
    }
    return v;
  }
}//end of class
```

Logical Flow

1. Using the `ReportManager` user interface (Figure 33.3), whenever a user selects a particular department and clicks on the `OK` button, the `Report-Manager` undergoes a change in its internal state (i.e., the value of its instance variable `department` changes).
2. As soon as the new state is set, the `ReportManager` invokes the `refreshData(Observable)` method on both the currently registered `MonthlyReport` and the `YTDChart` objects.
3. As part of `refreshData` method, both the report objects:
   a. Check to make sure that the subject that invoked the `refreshData` method is in fact the same Subject instance they have registered with. This is to prevent the observers from responding to unintended calls.

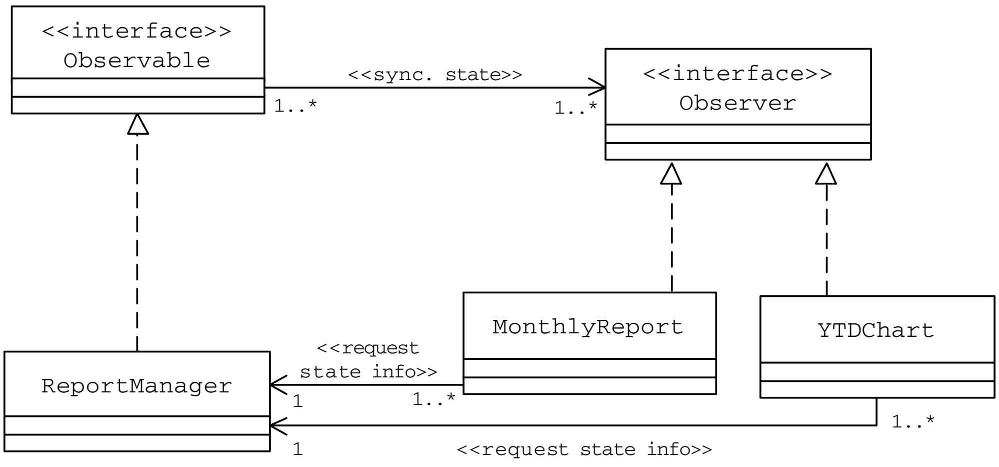
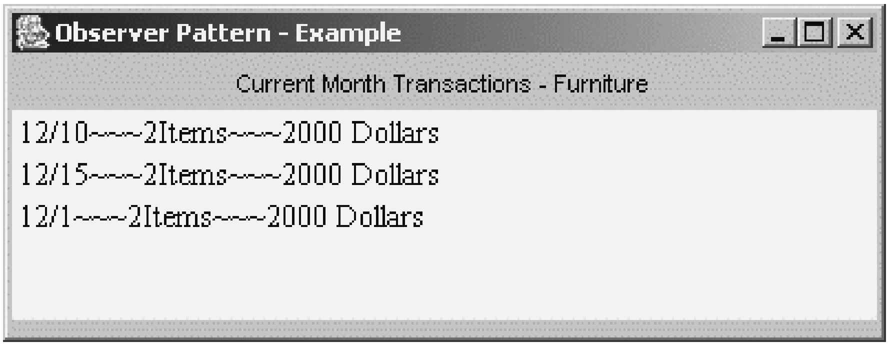


Figure 33.5 Example Application: Class Association

b. Query the `ReportManager` for its current state using the `getDepartment` method.
c. Retrieve appropriate data from the data file for display (Figures 33.6 and 33.7).

The sequence diagram in Figure 33.8 shows the communication between different objects when the application is run.

Whenever the state change logic implementation of the `ReportManager` changes, none of the observers will be affected. Similarly, when a new observer is added, the `ReportManager` class does not need to be changed.



Observer Pattern - Example
Current Month Transactions - Furniture

```
12/10-----2Items-----2000 Dollars
12/15-----2Items-----2000 Dollars
12/1-----2Items-----2000 Dollars
```

Figure 33.6 `MonthlyReport` View

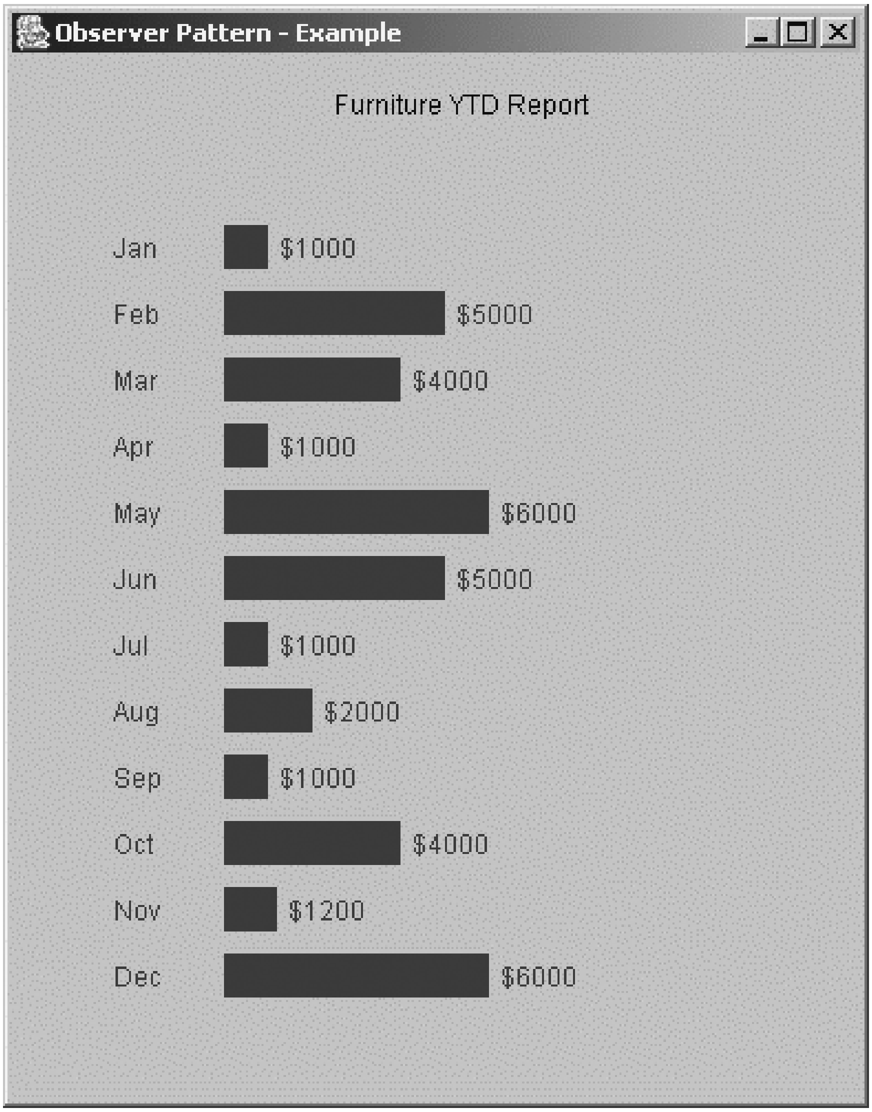


Figure 33.7 `YTDChart` **View**

PRACTICE QUESTIONS

1. Provide an implementation for the `unRegister` method of the `Report-Manager` class.
2. In general, it could lead to different problems if an observer changes the state of the subject (directly or indirectly) while attempting to update its state as part of its `refreshData(Observable)` method. Think of different ways of handling a scenario like this.

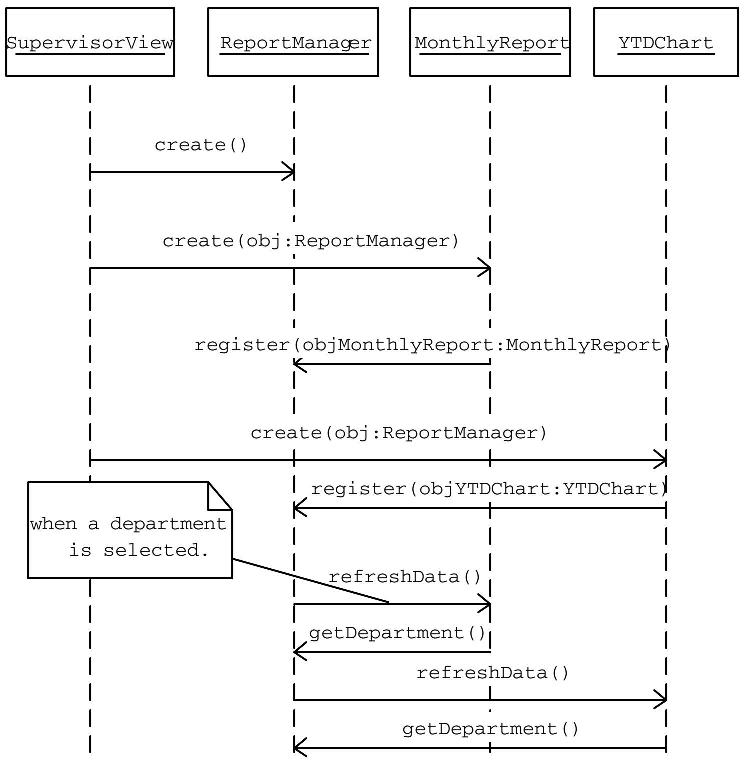

SupervisorView
ReportManager
MonthlyReport
YTDChart
create()
create(obj:ReportManager)
register(objMonthlyReport:MonthlyReport)
create(obj:ReportManager)
register(objYTDChart:YTDChart)
when a department is selected.
refreshData()
getDepartment()
refreshData()
getDepartment()

Figure 33.8 Application Message Flow

3. Design and implement an application for searching a jobs database that:
   - Allows a user to select a specific software skill and the number of years of experience.
   - Displays a list of all jobs that require the specific skill and the experience selected in one window with details.
   - Displays a list of all candidates with the specific skill and the experience selected in a third window.
4. Design and implement an application for monitoring and reporting different events with the following functionality:
   a. Whenever an event occurs, it is first sent to an `EventManager` object which functions as a publisher.
   b. Whenever the `EventManager` receives an event, it stores it to the database and sends notifications to the following three objects to take necessary action:
      i. An `AlertSender` object that sends notifications (e-mail or page) to different users depending on the event that occurred.
      ii. Two reporting objects that display the event data in different formats.