14

BUILDER

This pattern was previously described in GoF95.

DESCRIPTION

In general, object construction details such as instantiating and initializing the components that make up the object are kept within the object, often as part of its constructor. This type of design closely ties the object construction process with the components that make up the object. This approach is suitable as long as the object under construction is simple and the object construction process is definite and always produces the same representation of the object.

This design may not be effective when the object being created is complex and the series of steps constituting the object creation process can be implemented in different ways producing different representations of the object. Because different implementations of the construction process are all kept within the object, the object can become bulky (construction bloat) and less modular. Subsequently, adding a new implementation or making changes to an existing implementation requires changes to the existing code.

Using the Builder pattern, the process of constructing such an object can be designed more effectively. The Builder pattern suggests moving the construction logic out of the object class to a separate class referred to as *a builder* class. There can be more than one such builder class each with different implementation for the series of steps to construct the object. Each such builder implementation results in a different representation of the object. This type of separation reduces the object size. In addition:

- The design turns out to be more modular with each implementation contained in a different builder object.
- Adding a new implementation (i.e., adding a new builder) becomes easier.
- The object construction process becomes independent of the components that make up the object. This provides more control over the object construction process.

In terms of implementation, each of the different steps in the construction process can be declared as methods of a common interface to be implemented by different concrete builders. Figure 14.1 shows the resulting builder class hierarchy.

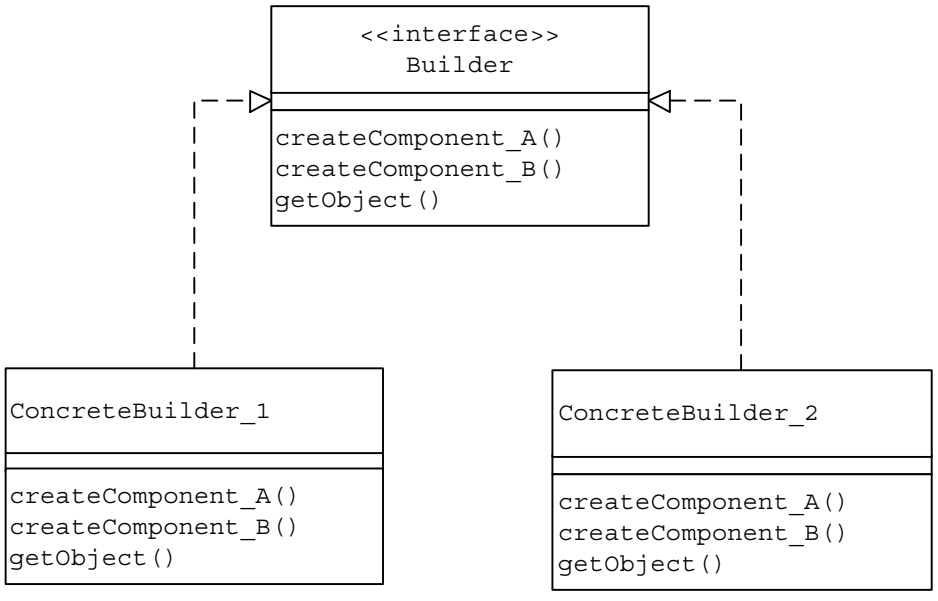

Figure 14.1 Generic `Builder` Class Hierarchy

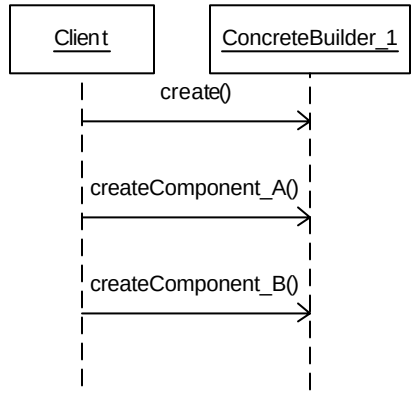

Figure 14.2 `Client/Builder` Direct Interaction

A client object can create an instance of a concrete builder and invoke the set of methods required to construct different parts of the final object. Figure 14.2 shows the corresponding message flow.

This approach requires every client object to be aware of the construction logic. Whenever the construction logic undergoes a change, all client objects need to be modified accordingly. The Builder pattern introduces another level of separation that addresses this problem. Instead of having client objects invoke different builder methods directly, the Builder pattern suggests using a dedicated object referred to as a *Director*, which is responsible for invoking different builder

methods required for the construction of the final object. Different client objects can make use of the Director object to create the required object. Once the object is constructed, the client object can directly request from the builder the fully constructed object. To facilitate this process, a new method `getObject` can be declared in the common Builder interface to be implemented by different concrete builders.

The new design eliminates the need for a client object to deal with the methods constituting the object construction process and encapsulates the details of how the object is constructed from the client. Figure 14.3 shows the association between different classes.

The interaction between the client object, the Director and the Builder objects can be summarized as follows:

- The client object creates instances of an appropriate concrete Builder implementer and the Director. The client may use a factory for creating an appropriate Builder object.
- The client associates the Builder object with the Director object.
- The client invokes the `build` method on the Director instance to begin the object creation process. Internally, the Director invokes different Builder methods required to construct the final object.
- Once the object creation is completed, the client invokes the `getObject` method on the concrete Builder instance to get the newly created object. Figure 14.4 shows the overall message flow.

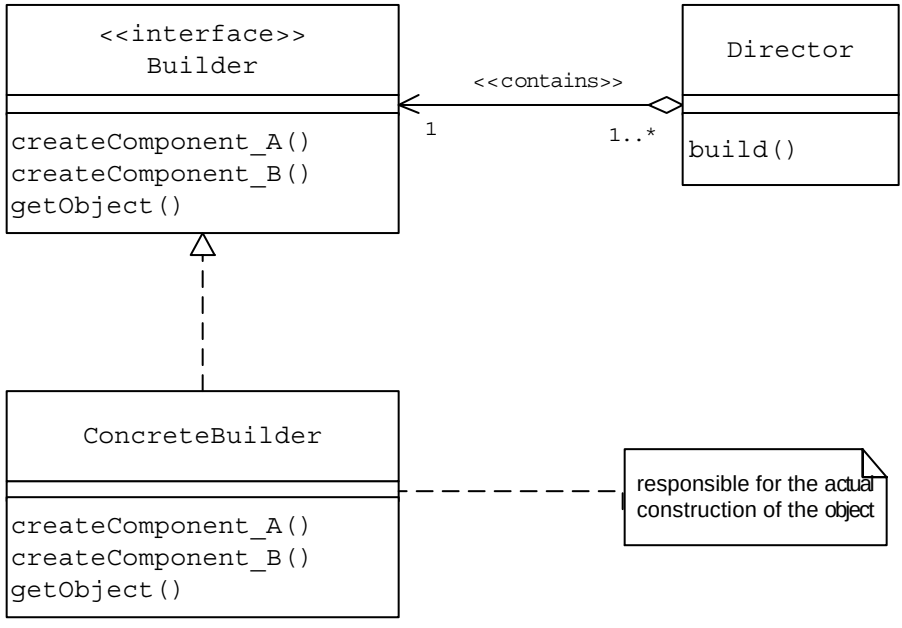

Figure 14.3 Class Association

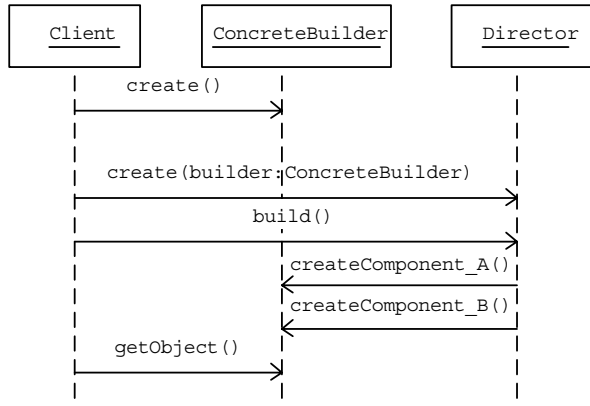

Figure 14.4 Object Creation When the Builder Pattern Is Applied

EXAMPLE I

A typical online job site maintains employer-, candidate- and jobs-related data. Let us build an application using the Builder pattern that displays the necessary user interface to allow a user to search for different employers and candidates in the database. For simplicity, let us consider only three fields for each search, which users can use to specify the search criteria.

- Employer Search
  - Name
  - City
  - Membership Renewal Date
- Candidate Search
  - Name
  - Experience (minimum number of years)
  - Skill Set

The required user interface (UI) for each of these searches requires a different combination of UI controls. In terms of implementation, the required set of UI controls can be placed in a `JPanel` container. The Builder pattern can be used in this case with different builder objects constructing the `JPanel` object with the necessary UI controls and initializing them appropriately.

Applying the Builder pattern, let us define the common builder interface in the form of an abstract `UIBuilder` class as in Listing 14.1.

Let us define two concrete subclasses (Figure 14.5 and Listing 14.2) of the `UIBuilder` class with responsibilities as listed in Table 14.1. These subclasses act as concrete builder classes.

A Side Note …

For simplicity, the `getSQL` method in both the `EmpSrchBuilder` the `CandSrchBuilder` is implemented to create the SQL statement as a string by simply

Listing 14.1 Abstract `UIBuilder` Class

```
public abstract class UIBuilder {
  protected JPanel searchUI;
  //add necessary UI controls and initialize them
  public abstract void addUIControls();
  public abstract void initialize();
  //return the SELECT sql command for the specified criteria
  public abstract String getSQL();
  //common to all concrete builders.
  //returns the fully constructed search UI
  public JPanel getSearchUI() {
    return searchUI;
  }
}
```

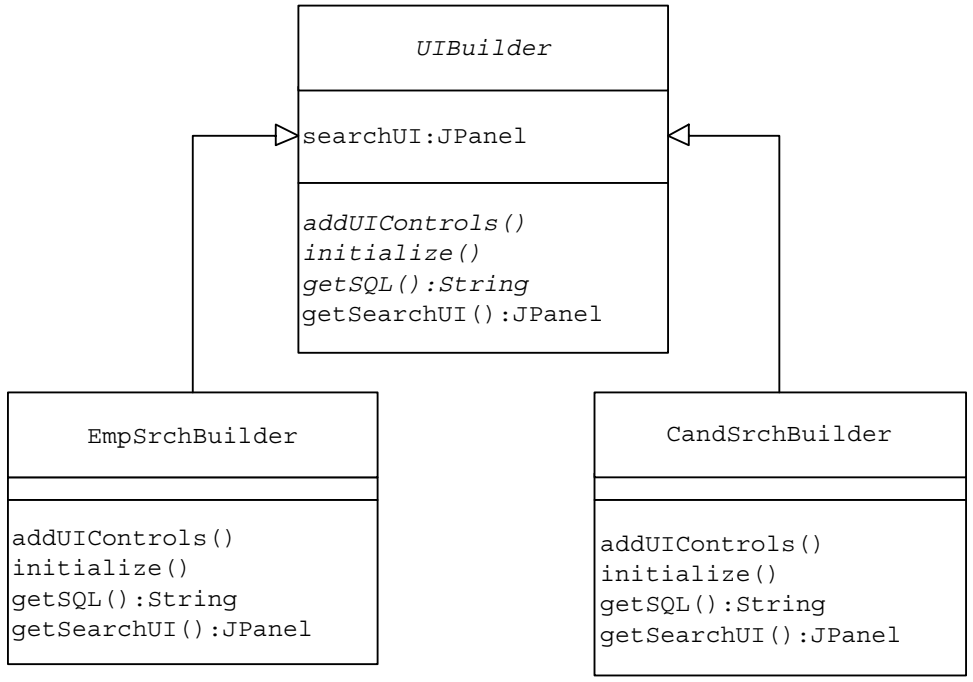

Figure 14.5 `UIBuilder`: Class Hierarchy

including the user input values, without any validations, in the SQL string. This type of implementation is likely to introduce an SQL injection problem. SQL injection is a technique that enables a malicious user to execute unauthorized

Listing 14.2 `UIBuilder` Concrete Subclasses

```
class EmpSrchBuilder extends UIBuilder {
          …
          …
 public void addUIControls() {
   searchUI = new JPanel();
   JLabel lblUserName = new JLabel("Name :");
   JLabel lblCity = new JLabel("City:");
   JLabel lblRenewal = new JLabel("Membership Renewal :");
   GridBagLayout gridbag = new GridBagLayout();
   searchUI.setLayout(gridbag);
   GridBagConstraints gbc = new GridBagConstraints();
   searchUI.add(lblUserName);
   searchUI.add(txtUserName);
   searchUI.add(lblCity);
   searchUI.add(txtCity);
   searchUI.add(lblRenewal);
   searchUI.add(txtRenewal);
          …
          …
   gbc.gridx = 0;
   gbc.gridy = 0;
   gridbag.setConstraints(lblUserName, gbc);
   gbc.gridx = 0;
   gbc.gridy = 1;
   gridbag.setConstraints(lblCity, gbc);
   gbc.gridx = 0;
   gbc.gridy = 2;
   gridbag.setConstraints(lblRenewal, gbc);
          …
          …
 }
 public void initialize() {
   Calendar cal = Calendar.getInstance();
   cal.setTime(new java.util.Date());
   txtUserName.setText("Enter UserName Here");
   txtRenewal.setText((cal.get(Calendar.MONTH) + 1) + "/" +
                      cal.get(Calendar.DATE) + "/" +
                      cal.get(Calendar.YEAR));
 }
```

(continued)

Listing 14.2 `UIBuilder` Concrete Subclasses (Continued)

```
  public String getSQL() {
    return ("Select * from Employer where Username='" +
             txtUserName.getText() + "'" + " and City='" +
             txtCity.getText() + "' and DateRenewal='" +
             txtRenewal.getText() + "'");
  }
}
class CandSrchBuilder extends UIBuilder {
          …
          …
  public void addUIControls() {
    searchUI = new JPanel();
    JLabel lblUserName = new JLabel("Name :");
    JLabel lblExperienceRange =
      new JLabel("Experience(min Yrs.):");
    JLabel lblSkill = new JLabel("Skill :");
    cmbExperience.addItem("<5");
    cmbExperience.addItem(">5");
    GridBagLayout gridbag = new GridBagLayout();
    searchUI.setLayout(gridbag);
    GridBagConstraints gbc = new GridBagConstraints();
    gbc.anchor = GridBagConstraints.WEST;
    searchUI.add(lblUserName);
    searchUI.add(txtUserName);
    searchUI.add(lblExperienceRange);
    searchUI.add(cmbExperience);
    searchUI.add(lblSkill);
    searchUI.add(txtSkill);
          …
          …
```

(continued)

Listing 14.2 `UIBuilder` Concrete Subclasses (Continued)

```
        gbc.gridx = 0;
        gbc.gridy = 0;
        gridbag.setConstraints(lblUserName, gbc);
        gbc.gridx = 0;
        gbc.gridy = 1;
        gridbag.setConstraints(lblExperienceRange, gbc);
        gbc.gridx = 0;
        gbc.gridy = 2;
        gridbag.setConstraints(lblSkill, gbc);
              …
              …
      }
      public void initialize() {
        txtUserName.setText("Enter UserName Here");
        txtSkill.setText("Internet Tech");
      }
      public String getSQL() {
        String experience =
          (String) cmbExperience.getSelectedItem();
        return ("Select * from Candidate where Username='" +
                txtUserName.getText() + "' and Experience " +
                experience + " and Skill='" +
                txtSkill.getText() + "'");
      }
    }
```

SQL commands by taking advantage of poor or no input validation when an SQL statement is built as a string, using user input values.

A malicious user could enter something like `joe';delete * from Employer` into the `Username` field, which results in an SQL statement as follows:

```
Select * from Employer where Username='joe';'delete *
from employer…
```

Most commercial database servers treat this as a batch of SQL statements. The first occurrence of `';` terminates the first SQL command and the server attempts to execute the next SQL statement in the batch, which is `delete * from employer`.

In this manner, attackers can trick the program into executing whatever SQL statement they want. In a real-world application, prepared statements (with

Table 14.1 Responsibilities of `EmpSrchBuilder` and `CandSrchBuilder` Concrete Builder Classes

Builder	*Responsibility*
`EmpSrchBuilder`	• Builds a `JPanel` object with the necessary UI controls for the employer search • Initializes UI controls • Returns the fully constructed `JPanel` object as part of the `getSearchUI` method • Builds the required SQL select command and returns it as part of the `getSQL` method
`CandSrchBuilder`	• Builds a `JPanel` object with the necessary UI controls for the candidate search • Initializes UI controls • Returns the fully constructed `JPanel` object as part of the `getSearchUI` method • Builds the required SQL select command and returns it as part of the `getSQL` method

placeholders instead of textual parameter insertion) should be used and parameters should be examined for dangerous characters before being passed on to the database.

Back to the Example Application …

Let us define a Director class `UIDirector` as in Listing 14.3.

The `UIDirector` maintains an object reference of type `UIBuilder`. This `UIBuilder` object can be passed to the `UIDirector` as part of a call to its constructor. As part of the `build` method, the `UIDirector` invokes different `UIBuilder` methods on this object for constructing the `JPanel searchUI` object.

Listing 14.3 `UIDirector` Class

```
public class UIDirector {
  private UIBuilder builder;
  public UIDirector(UIBuilder bldr) {
    builder = bldr;
  }
  public void build() {
    builder.addUIControls();
    builder.initialize();
  }
}
```

The client SearchManager can be designed (Listing 14.5) such that:

- It displays the necessary UI to allow a user to select the type of the search. The initial display contains an empty panel for the display of the search criteria UI (Figure 14.6).
- When the user selects a search type, the client object creates an instance of an appropriate UIBuilder using a BuilderFactory factory object. The BuilderFactory getUIBuilder method (Listing 14.4):
  - Accepts the type of the search selected by the user as input.
  - Creates an appropriate UIBuilder object based on this input and returns the UIBuilder object to the client.
- The client creates a UIDirector object and configures it with the UIBuilder object created above.
- The client invokes the build method of the UIDirector to begin the UI panel construction process. The UIDirector invokes the set of UIBuilder methods required to construct the JPanel object with the necessary UI controls.
- The client invokes the getSearchUI method on the UIBuilder object to access the fully constructed JPanel object, which contains the necessary user interface controls to allow a user to specify the search criteria. The JPanel search criteria UI is then displayed in the main UI window. Figures 14.7 and 14.8 show the UI displays for the employer search and candidate search, respectively.
- Once the user enters the search criteria and clicks on the GetSQL button, the client invokes the getSQL method on the UIBuilder object. Different concrete UIBuilder subclasses display different UI controls and the SQL statement depends on the fields represented by these controls but the client does not have to deal with these differences. Each of the concrete UIBuilder objects hides these details from the client and provides the implementation for the getSQL method, taking into account the representation of the object it builds.

Listing 14.4 BuilderFactory Class

```
class BuilderFactory {
  public UIBuilder getUIBuilder(String str) {
    UIBuilder builder = null;
    if (str.equals(SearchManager.CANDIDATE_SRCH)) {
      builder = new CandSrchBuilder();
    } else if (str.equals(SearchManager.EMPLOYER_SRCH)) {
      builder = new EmpSrchBuilder();
    }
    return builder;
  }
}
```

Listing 14.5 The Client `SearchManager` Class

```
public class SearchManager extends JFrame {
            …
            …
  public void actionPerformed(ActionEvent e) {
            …
            …
    if (e.getActionCommand().equals(SearchManager.GET_SQL)) {
      manager.setSQL(builder.getSQL());
    }
    if (e.getSource() == manager.getSearchTypeCtrl()) {
      String selection = manager.getSearchType();
      if (selection.equals("") == false) {
        BuilderFactory factory = new BuilderFactory();
        //create an appropriate builder instance
        builder = factory.getUIBuilder(selection);
        //configure the director with the builder
        UIDirector director = new UIDirector(builder);
        //director invokes different builder
        //methods
        director.build();
        //get the final build object
        JPanel UIObj = builder.getSearchUI();
        manager.displayNewUI(UIObj);
      }
    }
  }
            …
            …
  public buttonHandler(SearchManager inManager) {
    manager = inManager;
  }
}
```

The database interaction is not included in this example to keep it simple. The final SQL statement is simply displayed in the UI (Figure 14.9).

The class association can be depicted as in Figure 14.10.

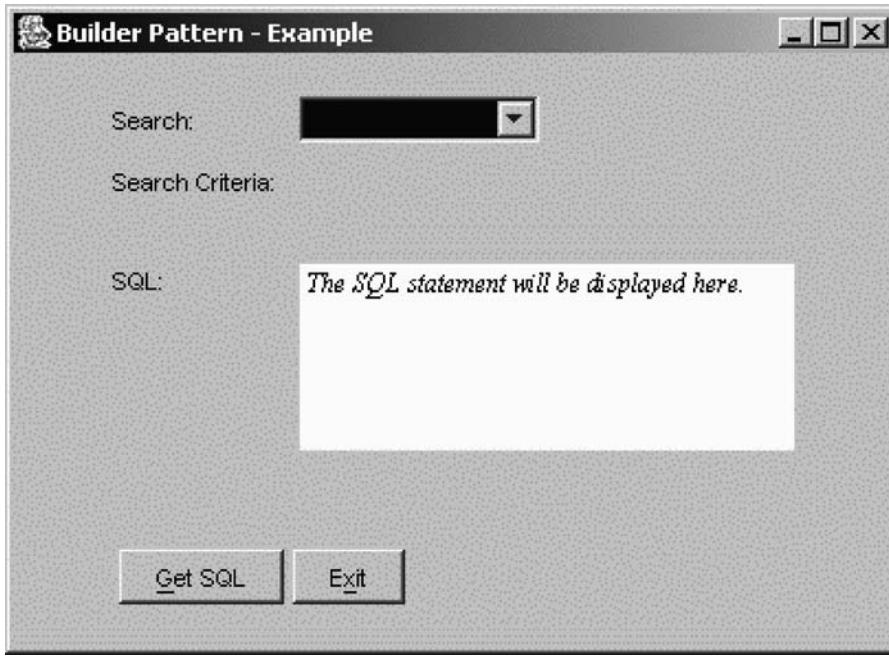

Figure 14.6 `SearchManager`: Initial UI Display

The `BuilderFactory` factory is not shown in the Figure 14.10 class diagram because it is not part of the Builder pattern implementation. The client `SearchManager` uses it only as a helper class.

The sequence diagram in Figure 14.11 shows the message flow when the user conducts an employer search.

EXAMPLE II

Let us design the following functionality for an online shopping site.

- A server side component receives the order information submitted by a user in the form of an XML string.
- The order XML is then parsed and validated to create an `Order` object.
- The `Order` object is finally saved to the disk.

A typical order XML record is shown follows:

```
<Order>
  <LineItems>
    <Item>
      <ID>100</ID>
      <Qty>1</Qty>
    </Item>
```

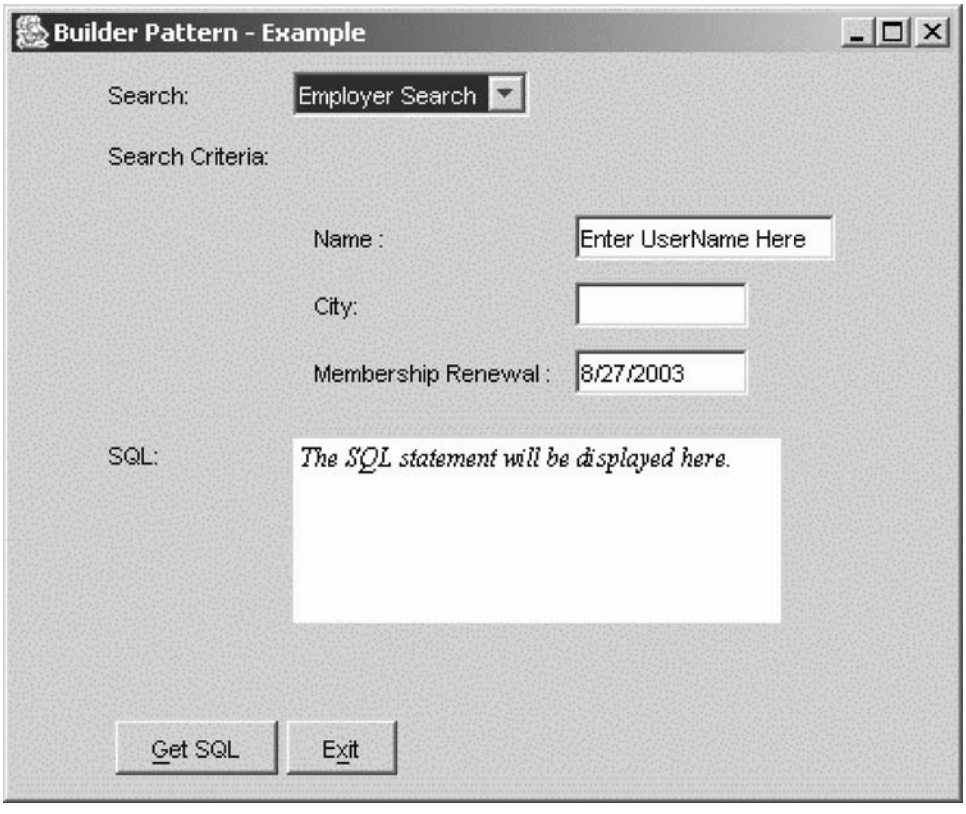

Figure 14.7 UI Display for the Employer Search

```
    <Item>
      <ID>200</ID>
      <Qty>2</Qty>
    </Item>
  </LineItems>
  <ShippingAddress>
    <Address1>101 Arrowhead Trail </Address1>
    <Address2> Suite 100</Address2>
    <City>Anytown</City>
    <State>OH</State>
    <Zip>12345</Zip>
  </ShippingAddress>
  <BillingAddress>
    <Address1>2669 Knox St </Address1>
    <Address2> Unit 444</Address2>
    <City>Anytown</City>
    <State>CA</State>
```

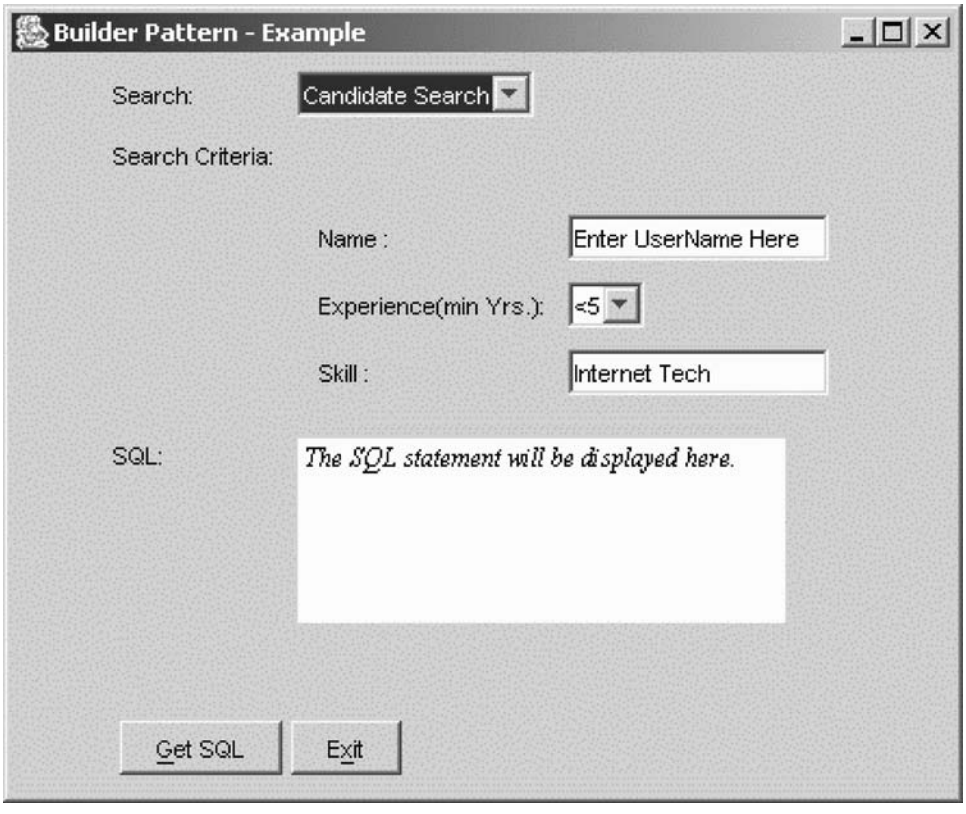

Figure 14.8 UI Display for the Candidate Search

```
      <Zip>56789</Zip>
    </BillingAddress>
  </Order>
```

Payment details are not included in XML in order to keep the example simple.

Let us consider three types of orders as in Table 14.2.

The class representation of a generic order can be designed as in Figure 14.12 with the required attributes and methods.

The `save` method can be used by different client objects to save the `Order` object to disk.

The series of steps required for the creation of an `Order` object can be summarized as follows:

- Parse the input XML string
- Validate the data
- Calculate the tax
- Calculate the shipping
- Create the actual object with:
  - Line items from the input XML string
  - Tax and shipping details calculated as per the details listed in Table 14.2

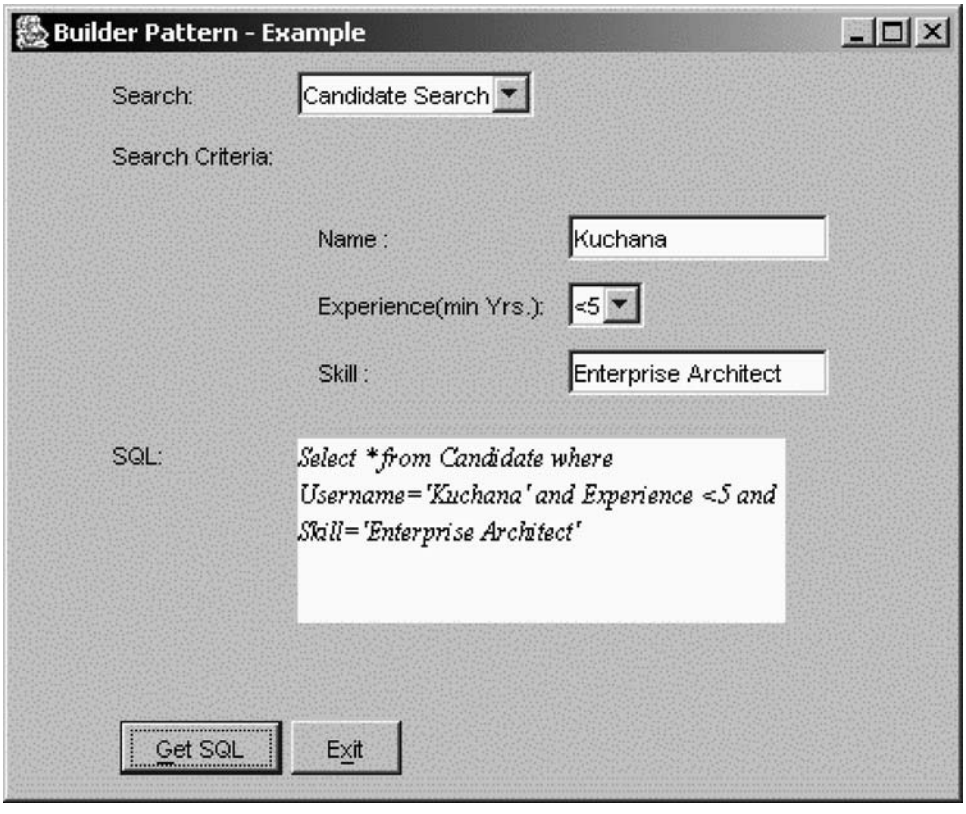

Figure 14.9 UI Display with SQL Statement Output

Let us design an interface `OrderBuilder` as in Figure 14.13 that declares the methods representing different steps in the `Order` object creation.

Because an order can exist in three different forms (California, Non-California or Overseas), let us define three concrete `OrderBuilder` implementers (Figure 14.14), where each implementer is responsible for the construction of a specific order type representation.

Each concrete `OrderBuilder` implementer can be designed to carry out the validations and tax and shipping calculation rules listed in Table 14.2 for the type of the `Order` object it constructs.

As a next step, let us define an `OrderDirector` as in Figure 14.15.

The `OrderDirector` contains an object reference of type `OrderBuilder`. The parse method is used internally by the `OrderDirector` to parse the input XML record. Figure 14.16 shows the overall association between different classes.

The server-side object that first receives the order XML acts as the client object in this case. The client makes use of the `OrderDirector` and concrete `OrderBuilder` implementer objects to create different representations of the `Order` object using the same construction process described as follows:

- The client first receives the order in the form of an XML record.

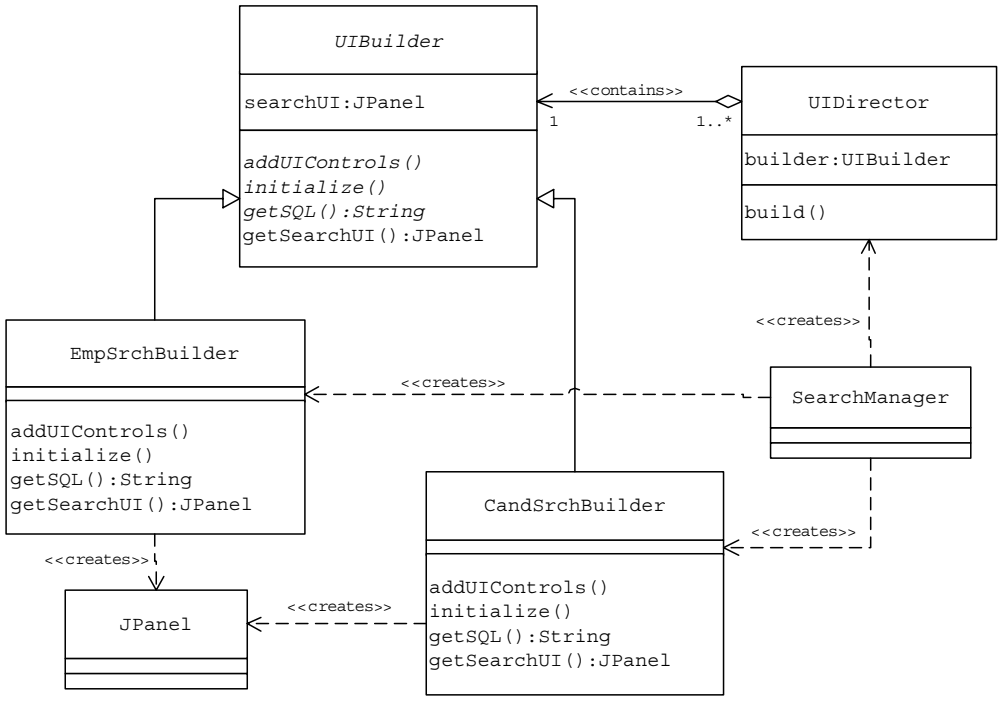

Figure 14.10 Class Association

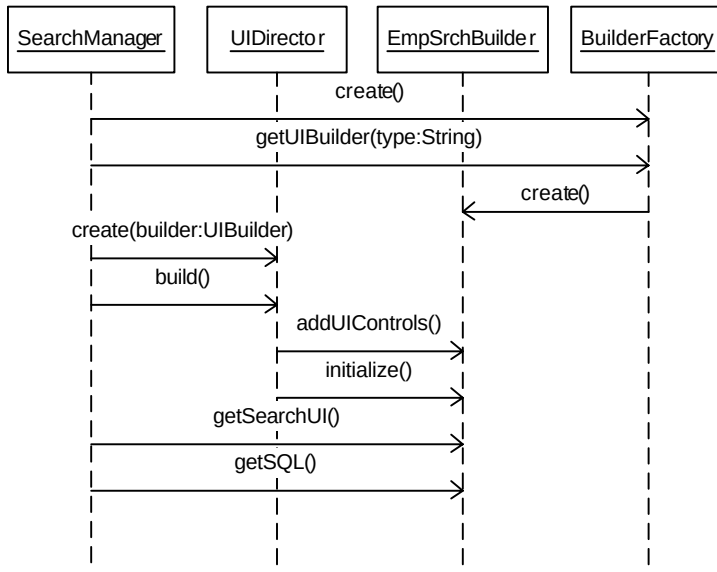

Figure 14.11 Message Flow

Table 14.2 Different Order Types

S. No	*Order Type*	*Details*
1	Overseas orders	• Orders from countries other than the United States. Additional shipping and handling is charged for these orders. • Overseas orders are accepted only if the order amount is greater than $100.
2	California orders	• U.S. orders with shipping address in California and are charged additional sales tax. • Orders with $100 or more order amount receive free regular shipping.
3	Non-California orders	• U.S. orders with shipping address not in California. Additional sales tax is not applicable. • Orders with $100 or more order amount receive free regular shipping.

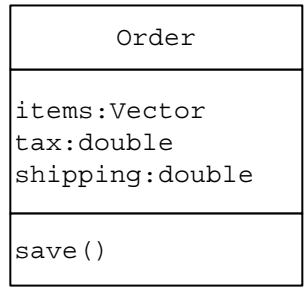

Figure 14.12 Generic Order Representation

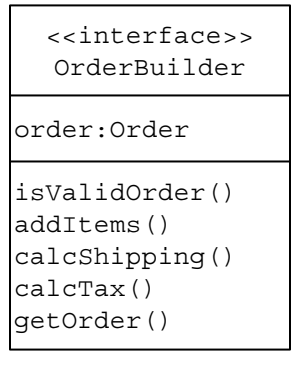

Figure 14.13 Builder Interface for Order Objects

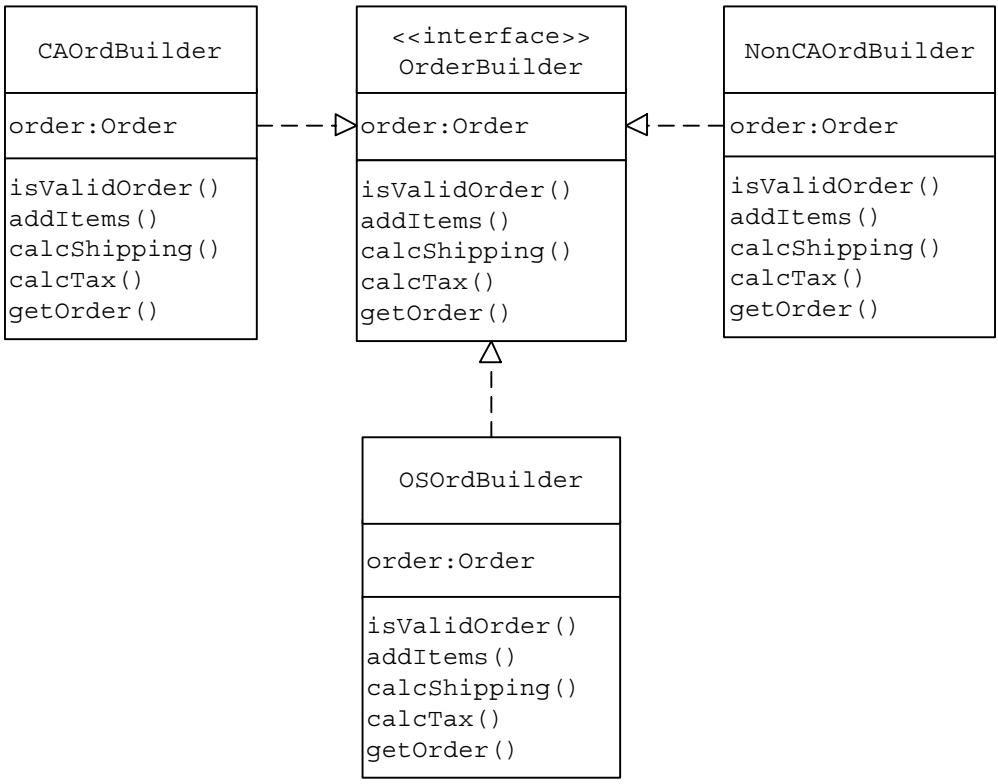

Figure 14.14 `OrderBuilder` Hierarchy

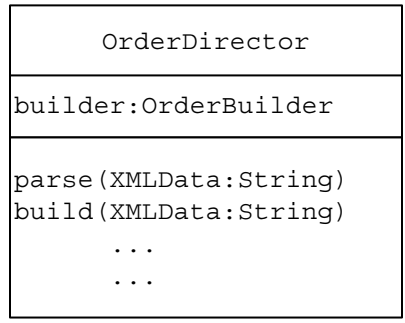

Figure 14.15 Director for the Creation of Order Objects

- The client creates an appropriate `OrderBuilder` object. It then instantiates the `OrderDirector`, passing the `OrderBuilder` object as a parameter.
- The client invokes the `build` method on the `OrderDirector`, passing the input XML data to initiate the `Order` object construction process.
  - If a problem is encountered during the construction process, a `BuilderException` is thrown. In general, error handling should be

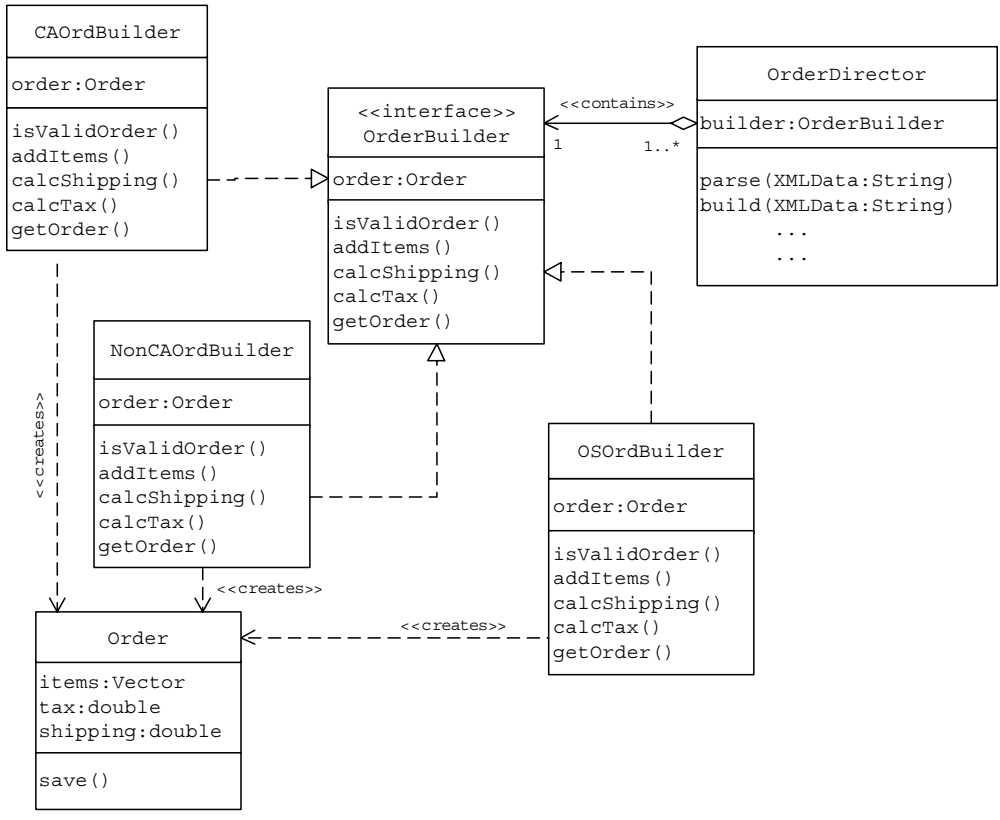

Figure 14.16 Class Association

done in a proper way with `Builders` to avoid the risk of having half-created objects lying around.

- The `OrderDirector` invokes different `OrderBuilder` methods to complete the construction of the `Order` object.
- The client calls the `getOrder` method of the `OrderBuilder` to get the final constructed `Order` object.
- The client can invoke the `save` method on the returned `Order` object to save the order to the disk.

EXAMPLE III

Let us design the order handling functionality for a different type of an online shopping site that transmits orders to different order fulfilling companies based on the type of the goods ordered. Suppose that the group of order processing companies can be classified into three categories based on the format of the order information they expect to receive. These formats include comma-separated value (CSV), XML and a custom object. When the order information is transformed into one of these formats, appropriate header and footer information that is specific to a format needs to be added to the order data.

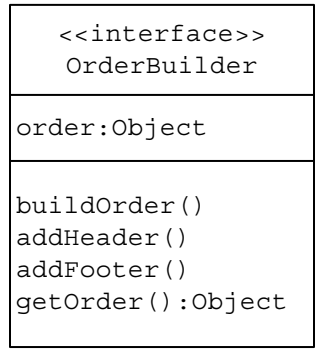

Figure 14.17 Builder Interface for Order Objects

The series of steps required for the creation of an `Order` object can be summarized as follows:

- Create the header specific to the format
- Add the order data
- Create the footer specific to the format

Let us design an interface `OrderBuilder` as in Figure 14.17 that declares the methods representing different steps in the `Order` object creation.

Because an order can exist in three different forms (CSV, XML and custom object), let us define three concrete `OrderBuilder` implementers as in Figure 14.18, where each implementer is responsible for the construction of a specific order representation.

Each concrete `OrderBuilder` implementer can be designed to implement the details of:

- Transforming input order information into a specific format
- Creating header or footer specific to the representation of the order being created

As a next step, let us define an `OrderDirector` as in Figure 14.19.

The `OrderDirector` contains an object reference of type `OrderBuilder`. Figure 14.20 shows the overall association between different classes.

Client objects can make use of the `OrderDirector` and concrete `OrderBuilder` implementer objects to create different representations of the `Order` object using the same construction process described as follows:

- The client creates an appropriate `OrderBuilder` object. It then instantiates the `OrderDirector`, passing the `OrderBuilder` object as a parameter.
- The client invokes the `build` method on the `OrderDirector`, passing the input order data to initiate the `Order` object construction process.

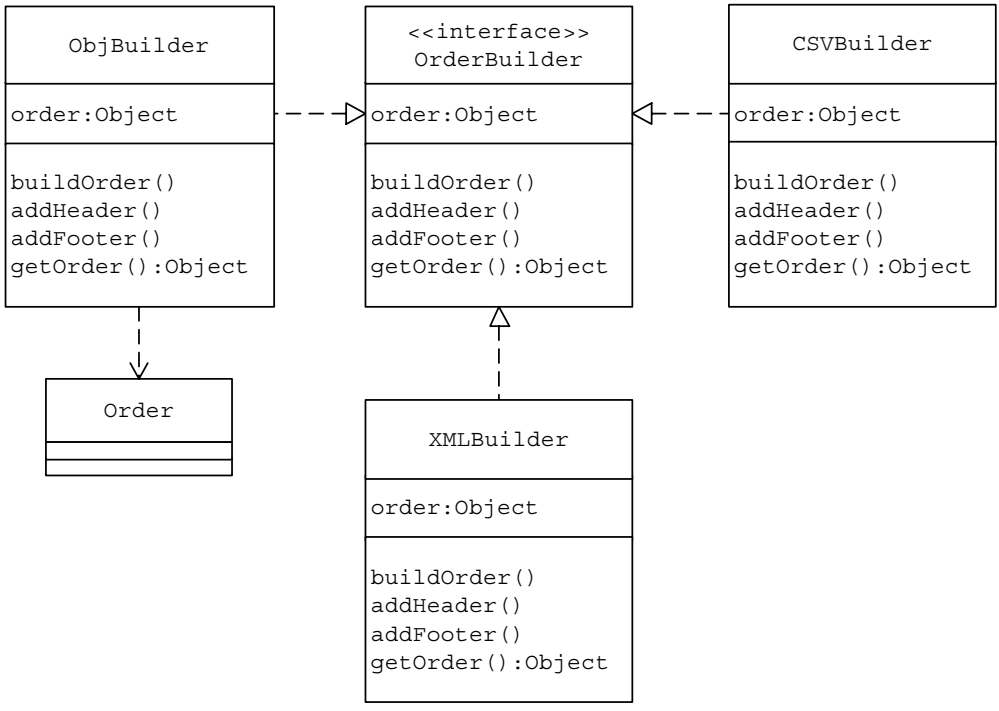

Figure 14.18 `OrderBuilder` Hierarchy

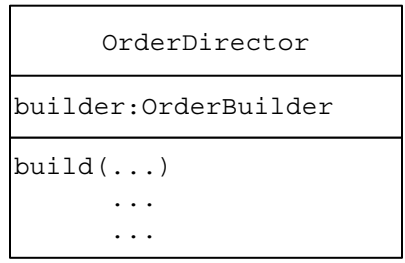

Figure 14.19 Director for the Creation of Order Objects

  - If a problem is encountered during the construction process, an exception `BuilderException` is thrown.
- The `OrderDirector` invokes the `buildOrder`, `addHeader` and `addFooter` `OrderBuilder` methods to complete the construction of the `Order` object.
- The client calls the `getOrder` method of the `OrderBuilder` to get the final constructed `Order` object.
- The client can transmit the order to an appropriate order handling company.

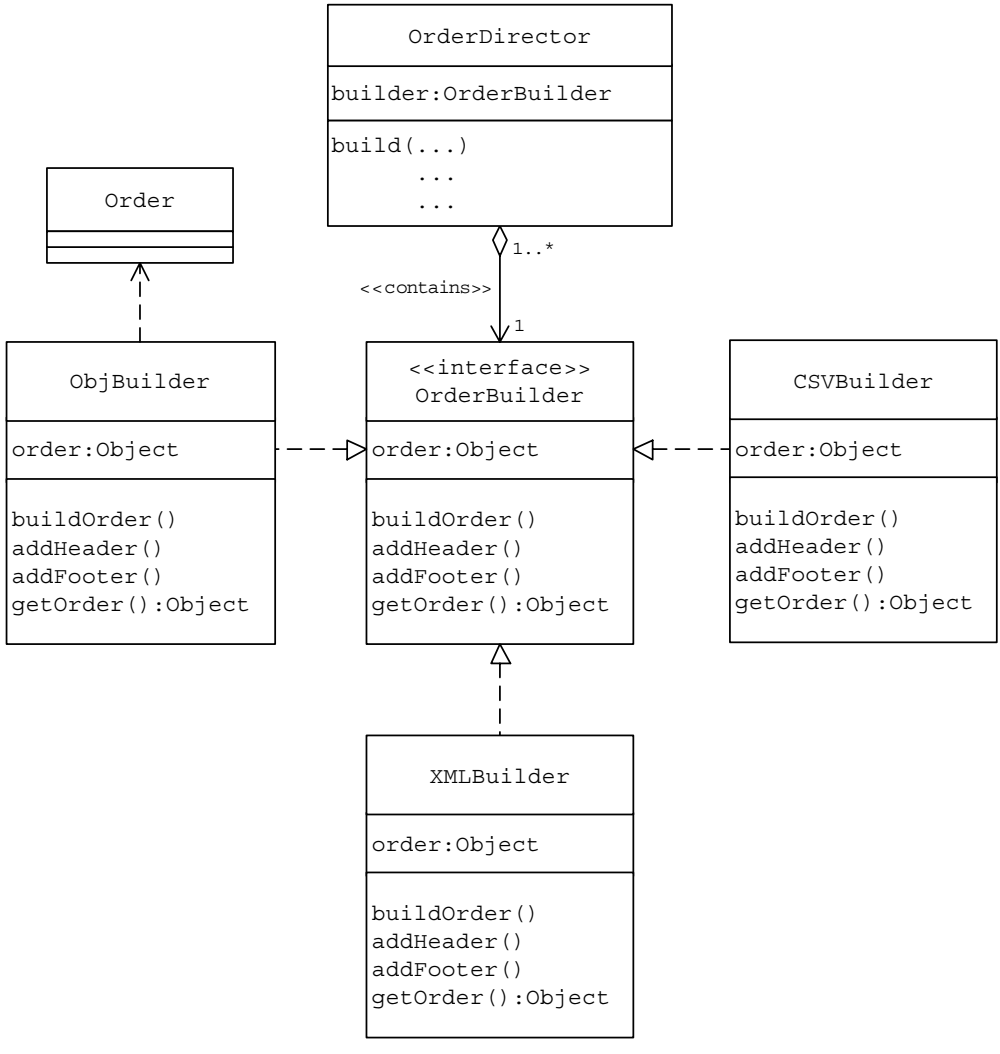

Figure 14.20 Class Association

PRACTICE QUESTIONS

1. Enhance the Example I application above to allow users to query on jobs as well. Create a new concrete builder to construct the necessary user interface.
2. Implement Examples II and III discussed above. Draw sequence diagrams to depict the message flow when the application is run.