22

FAÇADE

This pattern was previously described in GoF95.

DESCRIPTION

The Façade pattern deals with a subsystem of classes. A *subsystem* is a set of classes that work in conjunction with each other for the purpose of providing a set of related features (functionality). For example, an `Account` class, `Address` class and `CreditCard` class working together, as part of a subsystem, provide features of an online customer.

In real world applications, a subsystem could consist of a large number of classes. Clients of a subsystem may need to interact with a number of subsystem classes for their needs. This kind of direct interaction of clients with subsystem classes leads to a high degree of coupling between the client objects and the subsystem (Figure 22.1). Whenever a subsystem class undergoes a change, such as a change in its interface, all of its dependent client classes may get affected.

The Façade pattern is useful in such situations. The Façade pattern provides a higher level, simplified interface for a subsystem resulting in reduced complexity and dependency. This in turn makes the subsystem usage easier and more manageable.

A façade is a class that provides this simplified interface for a subsystem to be used by clients. With a `Façade` object in place, clients interact with the `Façade` object instead of interacting directly with subsystem classes. The `Façade` object takes up the responsibility of interacting with the subsystem classes. In effect, clients interface with the façade to deal with the subsystem. Thus the Façade pattern promotes a weak coupling between a subsystem and its clients (Figure 22.2).

From Figure 22.2, we can see that the `Façade` object decouples and shields clients from subsystem objects. When a subsystem class undergoes a change, clients do not get affected as before.

Even though clients use the simplified interface provided by the façade, when needed, a client will be able to access subsystem components directly through the lower level interfaces of the subsystem as if the `Façade` object does not exist. In this case, they will still have the same dependency/coupling issue as earlier.

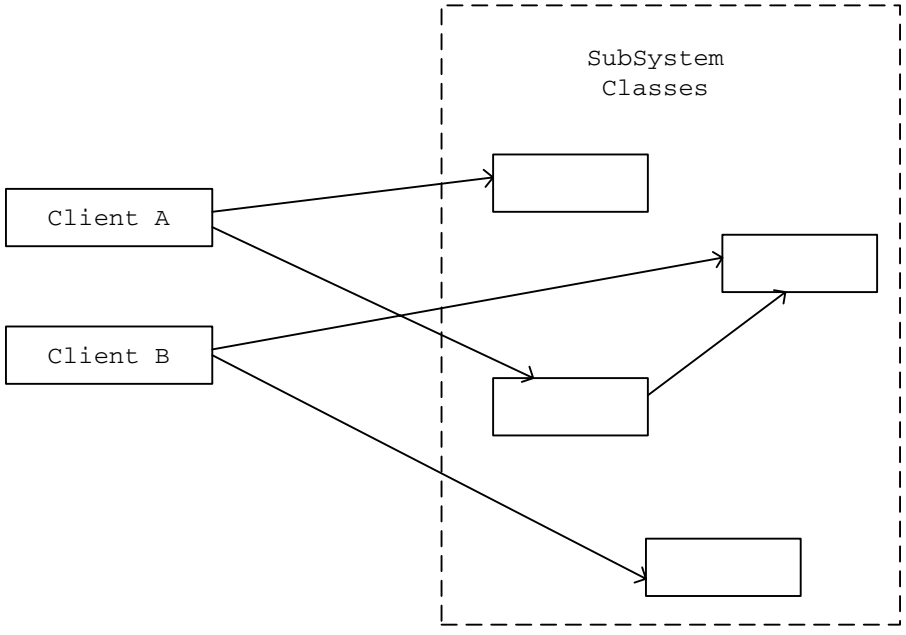

Figure 22.1 Client Interaction with Subsystem Classes before Applying the Façade Pattern

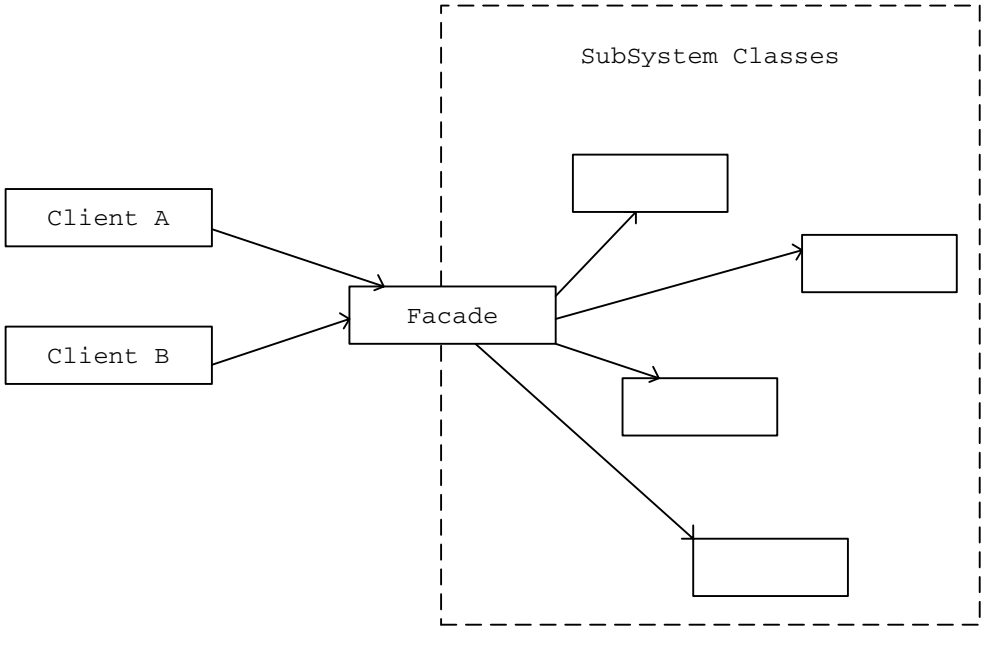

Figure 22.2 Client Interaction with Subsystem Classes after Applying the Façade Pattern

EXAMPLE

Let us build an application that:

- Accepts customer details (account, address and credit card details)
- Validates the input data
- Saves the input data to appropriate data files

Let us say that there are three classes — `Account`, `Address` and `CreditCard` — (Listing 22.1 through Listing 22.3) available in the system, each with its own methods for validating and saving the respective data (Figure 22.3).

Let us build a client `AccountManager` (Listing 22.4) that displays the user interface to a user to input the customer data.

When the client `AccountManager` is run, it displays the user interface shown in Figure 22.4.

In order to validate and save the input data, the client `AccountManager` would:

- Create `Account`, `Address` and `CreditCard` objects
- Validate the input data using these objects
- Save the input data using these objects

The sequence diagram in Figure 22.5 depicts the message flow between objects.

Applying the Façade pattern in this case can lead to a better design as it promotes low coupling between the client and the subsystem components (`Address`, `Account` and `CreditCard` classes in this case).

Applying the Façade pattern, let us define a `Façade` class `CustomerFacade` (Figure 22.6 and Listing 22.5) that offers a higher level, simplified interface to the subsystem consisting of customer data processing classes (`Address`, `Account` and `CreditCard`).

The `CustomerFacade` class offers a higher level business service in the form of the `saveCustomerData` method. Instead of interacting with each of the subsystem components directly, the client `AccountManager` can make use of the higher level, more simplified interface offered by the `CustomerFacade` object to validate and save the input customer data (Figure 22.7).

In the revised design, to validate and save the input customer data, the client needs to:

- Create or obtain an instance of the façade `CustomerFacade` class
- Send the data to be validated and saved to the `CustomerFacade` instance
- Invoke the `saveCustomerData` method on the `CustomerFacade` instance

The `CustomerFacade` handles the details of creating necessary subsystem objects and calling appropriate methods on those objects to validate and save the customer data. The client is no longer required to directly access any of the subsystem (`Account/Address/CreditCard`) objects.

Figure 22.8 shows the message flow in the revised design.

Listing 22.1 `Account` Class

```
public class Account {
  String firstName;
  String lastName;
  final String ACCOUNT_DATA_FILE = "AccountData.txt";
  public Account(String fname, String lname) {
    firstName = fname;
    lastName = lname;
  }
  public boolean isValid() {
    /*
      Let's go with simpler validation
      here to keep the example simpler.
    */
        …
        …
  }
  public boolean save() {
    FileUtil futil = new FileUtil();
    String dataLine = getLastName() + ”," + getFirstName();
    return futil.writeToFile(ACCOUNT_DATA_FILE, dataLine,
            true, true);
  }
  public String getFirstName() {
    return firstName;
  }
  public String getLastName() {
    return lastName;
  }
}
```

Listing 22.2 `Address` Class

```
public class Address {
  String address;
  String city;
  String state;
  final String ADDRESS_DATA_FILE = "Address.txt";
  public Address(String add, String cty, String st) {
    address = add;
    city = cty;
    state = st;
  }
  public boolean isValid() {
    /*
       The address validation algorithm
       could be complex in real-world
       applications.
       Let's go with simpler validation
       here to keep the example simpler.
     */
    if (getState().trim().length() < 2)
       return false;
    return true;
  }
  public boolean save() {
    FileUtil futil = new FileUtil();
    String dataLine = getAddress() + ”," + getCity() + ”," +
                      getState();
    return futil.writeToFile(ADDRESS_DATA_FILE, dataLine,
            true, true);
  }
  public String getAddress() {
    return address;
  }
  public String getCity() {
    return city;
  }
  public String getState() {
    return state;
  }
}
```

Listing 22.3 `CreditCard` Class

```
public class CreditCard {
  String cardType;
  String cardNumber;
  String cardExpDate;
  final String CC_DATA_FILE = "CC.txt";
  public CreditCard(String ccType, String ccNumber,
                    String ccExpDate) {
    cardType = ccType;
    cardNumber = ccNumber;
    cardExpDate = ccExpDate;
  }
  public boolean isValid() {
    /*
      Let's go with simpler validation
      here to keep the example simpler.
    */
    if (getCardType().equals(AccountManager.VISA)) {
      return (getCardNumber().trim().length() == 16);
    }
    if (getCardType().equals(AccountManager.DISCOVER)) {
      return (getCardNumber().trim().length() == 15);
    }
    if (getCardType().equals(AccountManager.MASTER)) {
      return (getCardNumber().trim().length() == 16);
    }
    return false;
  }
  public boolean save() {
    FileUtil futil = new FileUtil();
    String dataLine =
      getCardType() + ,"" + getCardNumber() + "," +
      getCardExpDate();
    return futil.writeToFile(CC_DATA_FILE, dataLine, true,
           true);
  }
```

(continued)

Listing 22.3 `CreditCard` Class (Continued)

```
    public String getCardType() {
      return cardType;
    }
    public String getCardNumber() {
      return cardNumber;
    }
    public String getCardExpDate() {
      return cardExpDate;
    }
  }
```

Figure 22.3 Subsystem Classes to Provide the Necessary Functionality to Validate and Save the Customer Data

Listing 22.4 Client `AccountManager` Class

```
public class AccountManager extends JFrame {
  public static final String newline = "\n";
  public static final String VALIDATE_SAVE = "Validate & Save";
          …
          …
  public AccountManager() {
    super(" Facade Pattern - Example ");
    cmbCardType = new JComboBox();
    cmbCardType.addItem(AccountManager.VISA);
    cmbCardType.addItem(AccountManager.MASTER);
    cmbCardType.addItem(AccountManager.DISCOVER);
          …
          …
    //Create buttons
    JButton validateSaveButton =
      new JButton(AccountManager.VALIDATE_SAVE);
          …
          …
  }
  public String getFirstName() {
    return txtFirstName.getText();
  }
          …
          …
}//End of class AccountManager
```

IMPORTANT NOTES

Here are few notes to consider while applying the Façade pattern:

- A façade should not be designed to provide any additional functionality.
- Never return subsystem components from `Façade` methods to clients. As an example, having a method as follows:

```
CreditCard getCreditCard()
```

  would expose the subsystem to clients and the application may not be able to realize the full benefits of using the Façade pattern.

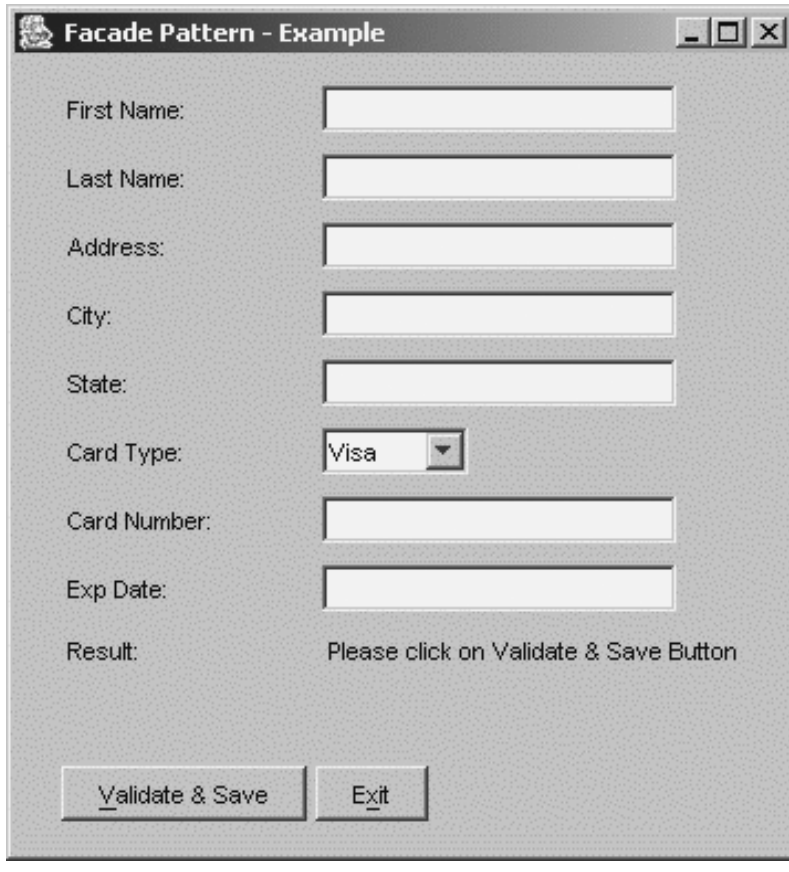

Figure 22.4 User Interface to Enter the Customer Data

- The objective of a façade is to provide a higher level interface and hence most preferably a typical `Façade` method should offer a higher level business service rather than performing a lower level individual task.

PRACTICE QUESTIONS

1. Design and implement a façade that can be used by different client objects to create a purchase request consisting of different line items, header data and other information.
2. Enhance the same `Façade` class to offer business services methods to:
   - Retrieve a purchase request from a database
   - Create a new purchase request by copying an existing purchase request

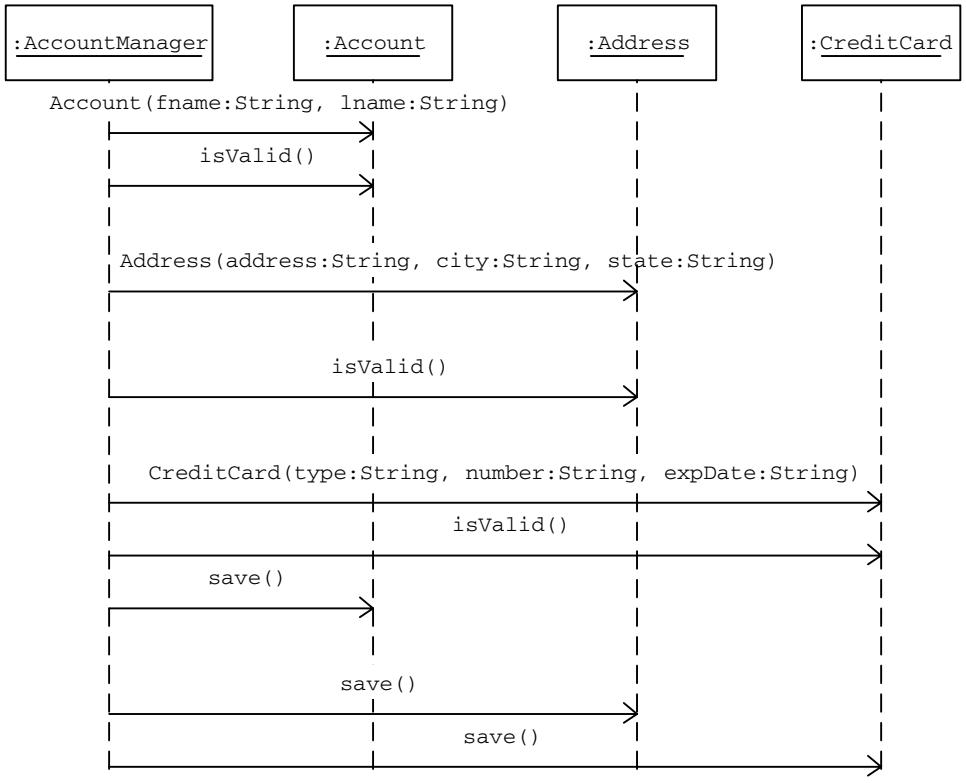

Figure 22.5 How a Client Would Normally Interact (Directly) with Subsystem Classes to Validate and Save the Customer Data

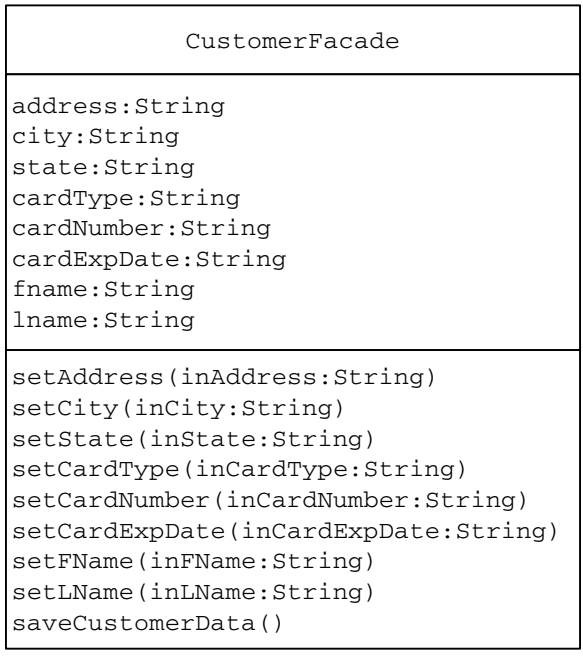

Figure 22.6 `Façade` Class to Be Used by the Client in the Revised Design

Listing 22.5 `CustomerFacade` Class

```
public class CustomerFacade {
  private String address;
  private String city;
  private String state;
  private String cardType;
  private String cardNumber;
  private String cardExpDate;
  private String fname;
  private String lname;
  public void setAddress(String inAddress) {
    address = inAddress;
  }
  public void setCity(String inCity) {
    city = inCity;
  }
  public void setState(String inState) {
    state = inState;
  }
  public void setFName(String inFName) {
    fname = inFName;
  }
  public void setLName(String inLName) {
    lname = inLName;
  }
  public void setCardType(String inCardType) {
    cardType = inCardType;
  }
  public void setCardNumber(String inCardNumber) {
    cardNumber = inCardNumber;
  }
  public void setCardExpDate(String inCardExpDate) {
    cardExpDate = inCardExpDate;
  }
  public boolean saveCustomerData() {
    Address objAddress;
    Account objAccount;
    CreditCard objCreditCard;
    /*
```

(continued)

Listing 22.5 `CustomerFacade` Class

```
        client is transparent from the following
        set of subsystem related operations.
      */
      boolean validData = true;
      String errorMessage = "";
      objAccount = new Account(fname, lname);
      if (objAccount.isValid() == false) {
        validData = false;
        errorMessage = "Invalid FirstName/LastName";
      }
      objAddress = new Address(address, city, state);
      if (objAddress.isValid() == false) {
        validData = false;
        errorMessage = "Invalid Address/City/State";
      }
      objCreditCard = new CreditCard(cardType, cardNumber,
                      cardExpDate);
      if (objCreditCard.isValid() == false) {
        validData = false;
        errorMessage = "Invalid CreditCard Info";
      }
      if (!validData) {
        System.out.println(errorMessage);
        return false;
      }
      if (objAddress.save() && objAccount.save() &&
          objCreditCard.save()) {
        return true;
      } else {
        return false;
      }
    }
  }
```

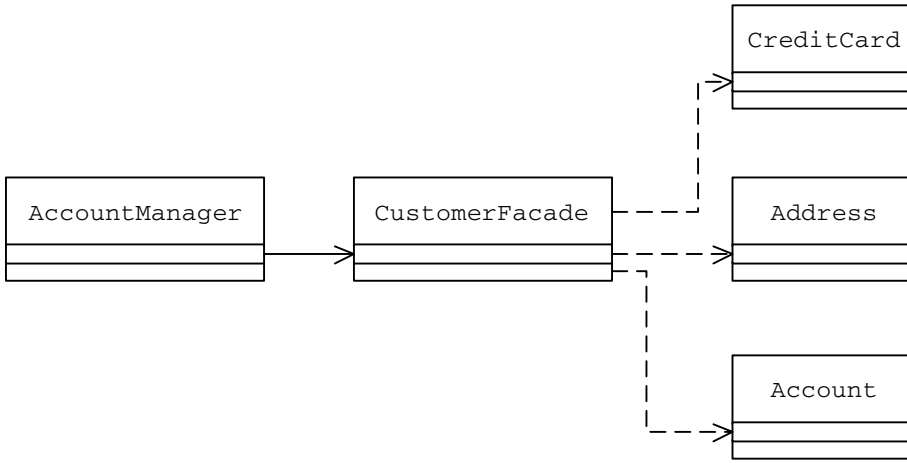

Figure 22.7 Class Association with the `Façade` Class in Place

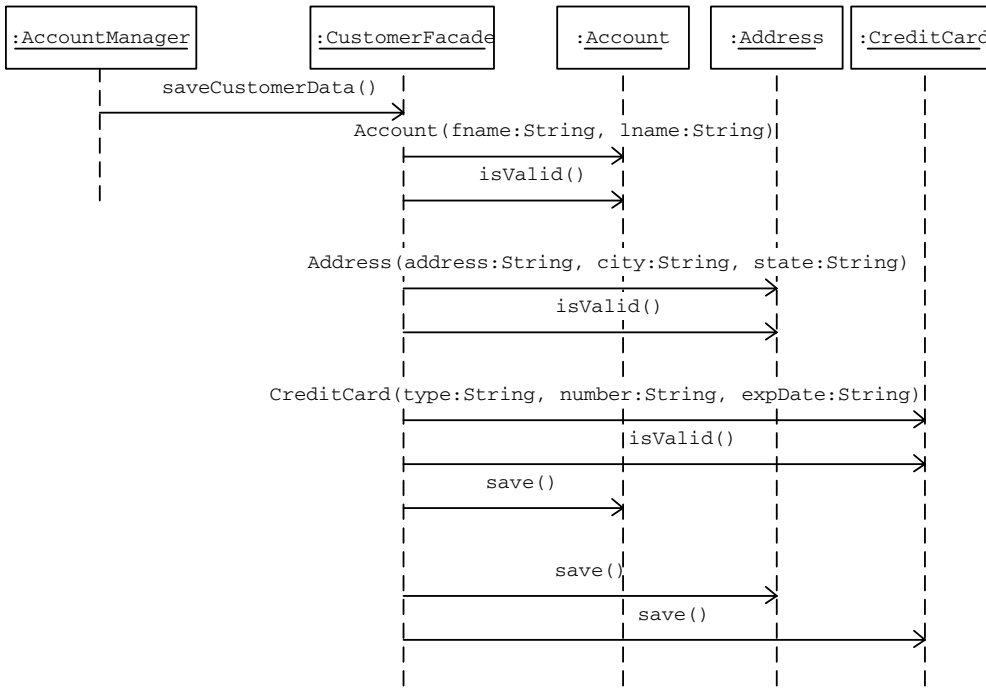

Figure 22.8 In the Revised Design, Clients Interact with the Façade Instance to Interface with the Subsystem