12

ABSTRACT FACTORY

This pattern was previously described in GoF95.

DESCRIPTION

During the discussion of the Factory Method pattern we saw that:

- In the context of a factory method, there exists a class hierarchy composed of a set of subclasses with a common parent class.
- A factory method is used when a client object knows when to create an instance of the parent class type, but does not know (or should not know) exactly which class from among the set of subclasses (and possibly the parent class) should be instantiated. Besides the class selection criteria, a factory method also hides any special mechanism required to instantiate the selected class.

The Abstract Factory pattern takes the same concept to the next level. In simple terms, an *abstract factory* is a class that provides an interface to produce a family of objects. In the Java programming language, it can be implemented either as an interface or as an abstract class.

In the context of an abstract factory there exist:

- Suites or families of related, dependent classes.
- A group of concrete factory classes that implements the interface provided by the abstract factory class. Each of these factories controls or provides access to a particular suite of related, dependent objects and implements the abstract factory interface in a manner that is specific to the family of classes it controls.

The Abstract Factory pattern is useful when a client object wants to create an instance of one of a suite of related, dependent classes without having to know which specific concrete class is to be instantiated. In the absence of an abstract factory, the required implementation to select an appropriate class (in other words, the class selection criterion) needs to be present everywhere such an instance is created. An abstract factory helps avoid this duplication by providing the necessary

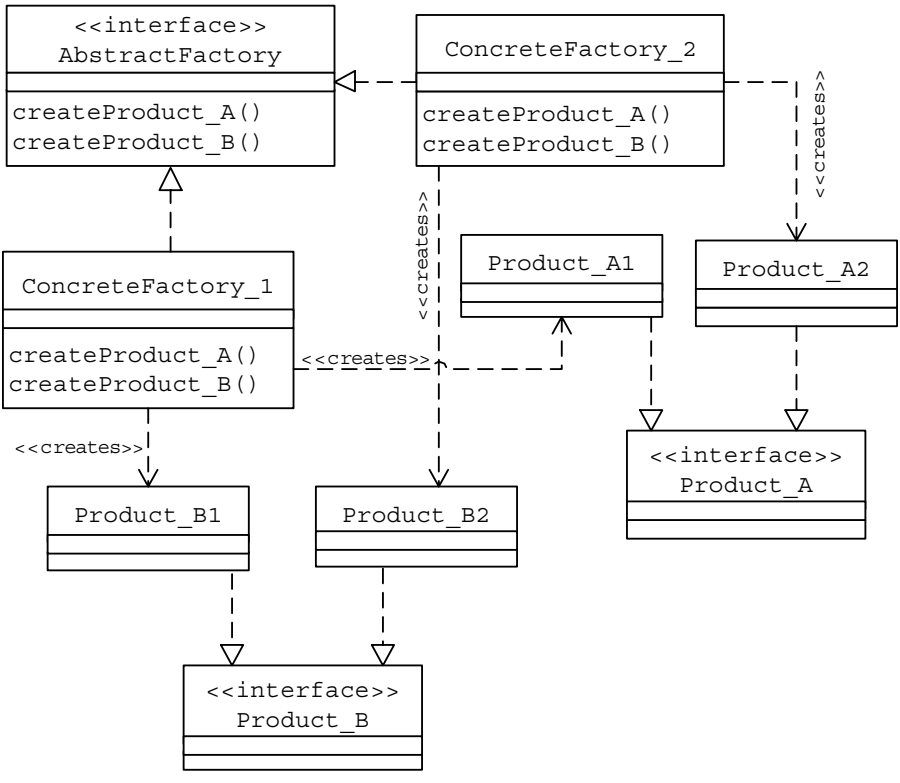

Figure 12.1 Generic Class Associations While Applying the Abstract Factory Pattern

interface for creating such instances. Different concrete factories implement this interface. Client objects make use of these concrete factories to create objects and, therefore, do not need to know which concrete class is actually instantiated. Figure 12.1 shows the generic class association when the Abstract Factory pattern is applied.

The abstract factory shown in the Figure 12.1 class diagram is designed as a Java interface with its implementers as concrete factories. In Java, an abstract factory can also be designed as an abstract class with its concrete subclasses as factories, where each factory is responsible for creating and providing access to the objects of a particular suite of classes.

ABSTRACT FACTORY VERSUS FACTORY METHOD

Abstract Factory is used to create groups of related objects while hiding the actual concrete classes. This is useful for plugging in a different group of objects to alter the behavior of the system. For each group or family, a concrete factory is implemented that manages the creation of the objects and the interdependencies and consistency requirements between them. Each concrete factory implements the interface of the abstract factory.

This situation often arises when designing a framework or a library, which needs to be kept extensible. One example is the JDBC (Java Database Connectivity)

driver system, where each driver contains classes that implement the `Connection`, the `Statement` and the `ResultSet` interfaces. The set of classes that the Oracle JDBC driver contains are different from the set of classes that the DB2 JDBC driver contains and they must not be mixed up. This is where the role of the factory comes in: It knows which classes belong together and how to create objects in a consistent way.

Factory Method is specifying a method for the creation of an object, thus allowing subclasses or implementing classes to define the concrete object. Abstract Factories are usually implemented using the Factory Method pattern. Another approach would be to use the Prototype pattern.

EXAMPLE I

Let us design an application to query the features of different types of vehicles. For simplicity, let us consider two types of vehicles: cars and SUVs. Further, a vehicle can be of either luxury or nonluxury category.

Let us define a common `Car` interface (Figure 12.2, Listing 12.1) that declares the interface to be implemented by different classes that represent different types of cars.

Let us design two classes (Listing 12.2) that implement the `Car` interface — `LuxuryCar` and `NonLuxuryCar` — representing luxury cars and nonluxury cars, respectively. Figure 12.2 shows the resulting class hierarchy.

A similar class hierarchy can be designed to represent SUVs (Figure 12.3, Listing 12.3). In Figure 12.3, the `SUV` interface declares the common interface to be offered by different classes that represent SUVs and its implementers — `LuxurySUV` and `NonLuxurySUV` — (Listing 12.4) represent luxury and nonluxury SUVs, respectively.

For simplicity, both the `Car` and the `SUV` class hierarchies are designed to offer only two basic operations to retrieve the details of a car or SUV. Together these class hierarchies contain two families of classes as listed in Table 12.1.

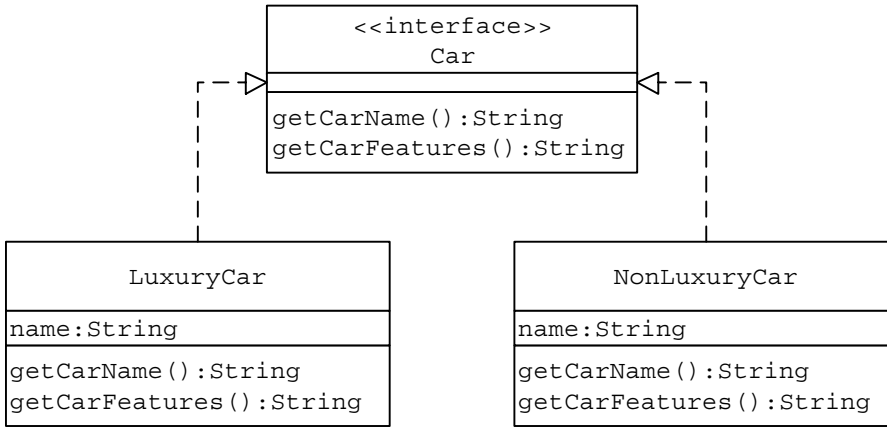

Figure 12.2 The `Car` Class Hierarchy

Listing 12.1 The Common Car Interface

```
public interface Car {
  public String getCarName();
  public String getCarFeatures();
}//End of class
```

Listing 12.2 Classes Representing Luxury and NonLuxury Cars

```
public class LuxuryCar implements Car {
  private String name;
  public LuxuryCar(String cName) {
    name = cName;
  }
  public String getCarName() {
    return name;
  }
  public String getCarFeatures() {
    return "Luxury Car Features ";
  };
}//End of class
public class NonLuxuryCar implements Car {
  private String name;
  public NonLuxuryCar(String cName) {
    name = cName;
  }
  public String getCarName() {
    return name;
  }
  public String getCarFeatures() {
    return "Non-Luxury Car Features ";
  };
}//End of class
```

An application object can make use of the services of different `Car/SUV` implementers (Table 12.1) to query the features of a car or SUV. If such an application object is to directly deal with different concrete `Car/SUV` classes, it needs to be aware of the existence of different concrete `Car/SUV` implementers. In addition, this approach results in the required implementation to select and instantiate an appropriate `Car/SUV` class to be present everywhere an application

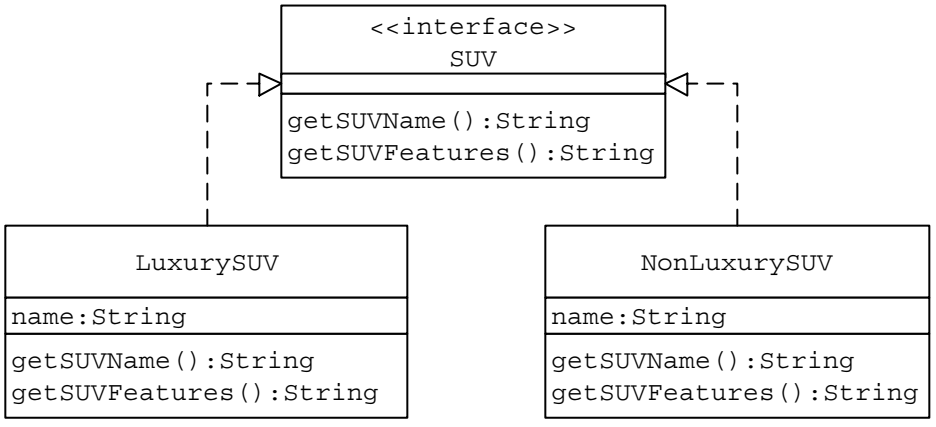

Figure 12.3 The SUV Class Hierarchy

Listing 12.3 The Common SUV Interface

```
public interface SUV {
  public String getSUVName();
  public String getSUVFeatures();
}//End of class
```

object needs to query the features of a car or SUV. Using the Abstract Factory pattern in such cases, the responsibility of selecting and instantiating an appropriate `Car/SUV` implementer can be moved out of application objects to a separate designated abstract factory class. The abstract factory class may simply declare the required interface, leaving the actual implementation details of class selection and instantiation to its implementers. This type of design eliminates the need for an application object to be aware of the existence of different concrete `Car/SUV` classes.

Applying the Abstract Factory pattern in this case, let us define an interface in the form of an abstract class `VehicleFactory` (Figure 12.4, Listing 12.5).

The `VehicleFactory` class acts as an abstract factory. A client object can use an instance of a concrete factory that implements the `VehicleFactory` interface to create objects representing vehicles of different types and categories without having to know the actual concrete class that needs to be instantiated. As we proceed with the design, we will create a utility method that can be used by different client objects to obtain an appropriate concrete factory (which implements the abstract factory interface).

As part of applying the Abstract Factory pattern, let us define two concrete factory classes — `LuxuryVehicleFactory` and `NonLuxuryVehicleFactory` – as concrete subclasses (Listing 12.6) of the `VehicleFactory` with responsibilities as detailed in Table 12.2. The class diagram in Figure 12.5 shows the resulting class hierarchy.

Listing 12.4 Classes Representing Luxury and NonLuxury SUVs

```
public class LuxurySUV implements SUV {
  private String name;
  public LuxurySUV(String sName) {
    name = sName;
  }
  public String getSUVName() {
    return name;
  }
  public String getSUVFeatures() {
    return "Luxury SUV Features ";
  };
}//End of class
public class NonLuxurySUV implements SUV {
  private String name;
  public NonLuxurySUV(String sName) {
    name = sName;
  }
  public String getSUVName() {
    return name;
  }
  public String getSUVFeatures() {
    return "Non-Luxury SUV Features ";
  };
}//End of class
```

Table 12.1 Families of Vehicle Classes

Family	*Member Classes*
Luxury	`LuxuryCar, LuxurySUV`
Nonluxury	`NonLuxuryCar, NonLuxurySUV`

Now, it also needs to be ensured that client objects do not have to know about the existence of these concrete factory classes. A client object should be provided with an appropriate factory object as needed.

To facilitate this, let us add a static method `getVehicleFactory(String type)` to the `VehicleFactory` abstract class (Figure 12.6). This new method can be implemented to return an appropriate concrete vehicle factory object

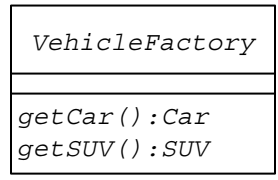

Figure 12.4 The Abstract Factory Class

Listing 12.5 Abstract `VehicleFactory` Class

```
public abstract class VehicleFactory {
  public static final String LUXURY_VEHICLE = "Luxury";
  public static final String NON_LUXURY_VEHICLE = "Non-Luxury";
  public abstract Car getCar();
  public abstract SUV getSUV();
          …
          …
}//End of class
```

Listing 12.6 Concrete Factory Subclasses of the Abstract `VehicleFactory` Class

```
public class LuxuryVehicleFactory extends VehicleFactory {
  public Car getCar() {
    return new LuxuryCar("L-C");
  }
  public SUV getSUV() {
    return new LuxurySUV("L-S");
  }
}//End of class
public class NonLuxuryVehicleFactory extends VehicleFactory {
  public Car getCar() {
    return new NonLuxuryCar("NL-C");
  }
  public SUV getSUV() {
    return new NonLuxurySUV("NL-S");
  }
}//End of class
```

Table 12.2 Different Concrete Vehicle Factories

Concrete Factory	*Responsibility*
`LuxuryVehicleFactory`	Responsible for creating instances of classes representing luxury vehicles
`NonLuxuryVehicleFactory`	Responsible for creating instances of classes representing nonluxury vehicles

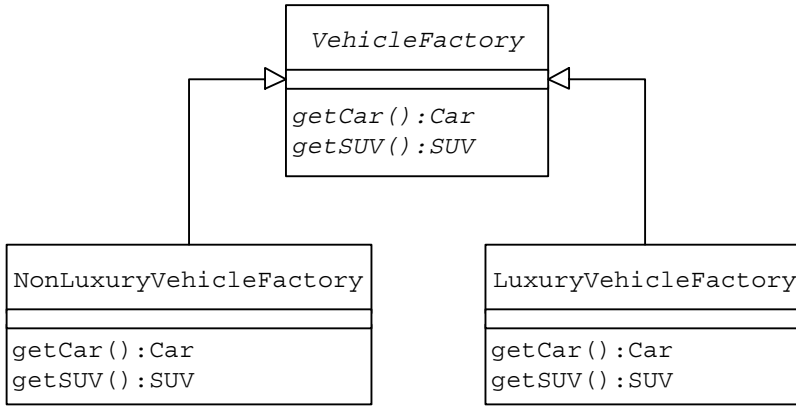

Figure 12.5 Abstract `VehicleFactory` Class Hierarchy

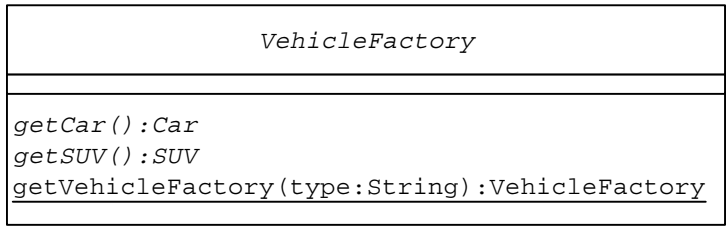

Figure 12.6 Revised Abstract Factory Class

(`LuxuryVehicleFactory` or `NonLuxuryVehicleFactory` in this case) to a calling client object (Listing 12.7).

The `getVehicleFactory(String type)` method does not need to be within the abstract factory as we designed in this example application. It can be in a different class altogether. In such a case, the abstract factory can be implemented as a Java interface instead of its current design as an abstract class.

Listing 12.7 Abstract `VehicleFactory` Class: Revised

```
public abstract class VehicleFactory {
            …
            …
  public static VehicleFactory getVehicleFactory(String type) {
    if (type.equals(VehicleFactory.LUXURY_VEHICLE))
      return new LuxuryVehicleFactory();
    if (type.equals(VehicleFactory.NON_LUXURY_VEHICLE))
      return new NonLuxuryVehicleFactory();
    return new LuxuryVehicleFactory();
  }
}//End of class
```

Let us see how a typical client object can make use of the class structure we have put together so far in this example.

The example client object `AutoSearchUI` (Listing 12.8) displays the necessary user interface as in Figure 12.7 for querying different vehicle features.

When the `Search` button is clicked after a vehicle category and type combination is selected:

1. The client `AutoSearchUI` invokes the static `getVehicleFactory(String type)` method on the abstract `VehicleFactory` class.
2. The `getVehicleFactory` method creates an appropriate factory object and returns it as an object of `VehicleFactory` type.
3. The client `AutoSearchUI` does not need to know the existence of any concrete vehicle factory class. It simply invokes the required abstract vehicle factory method such as `getCar` or `getSUV` on the returned factory instance.
4. The factory object internally creates an instance of an appropriate class from among the family of classes it controls (`LuxuryCar/LuxurySUV` or `NonLuxuryCar/NonLuxurySUV`) and returns it as an object of `Car/SUV` type.
5. The client `AutoSearchUI` does not need to be aware of the existence of different concrete `Car/SUV` classes. It simply invokes methods declared by the `Car/SUV` interface such as `getName` or `getFeatures` on the object returned in Step 4 above.

The sequence diagram in Figure 12.8 depicts the message exchange between objects when the client `AutoSearchUI` uses `VehicleFactory` (i.e., the abstract factory) to retrieve luxury car details.

Listing 12.8 Client `AutoSearchUI` Class

```
public class AutoSearchUI extends JFrame {
          …
          …
  public void actionPerformed(ActionEvent e) {
    String searchResult = null;
     if (e.getActionCommand().equals(AutoSearchUI.EXIT)) {
       System.exit(1);
    }
    if (e.getActionCommand().equals(AutoSearchUI.SEARCH)) {
      //get input values
      String vhCategory =
        objAutoSearchUI.getSelectedCategory();
      String vhType = objAutoSearchUI.getSelectedType();
      //get one of Luxury or NonLuxury vehicle factories
      VehicleFactory vf =
        VehicleFactory.getVehicleFactory(vhCategory);
      if (vhType.equals(AutoSearchUI.CAR)) {
        Car c = vf.getCar();
        searchResult =
          "Name: " + c.getCarName() + " Features: " +
          c.getCarFeatures();
      }
      if (vhType.equals(AutoSearchUI.SUV)) {
        SUV s = vf.getSUV();
        searchResult =
          "Name: " + s.getSUVName() + " Features: " +
          s.getSUVFeatures();
      }
      objAutoSearchUI.setResult(searchResult);
    }
  }
          …
          …
}
```

Figure 12.7 Vehicle Query User Interface

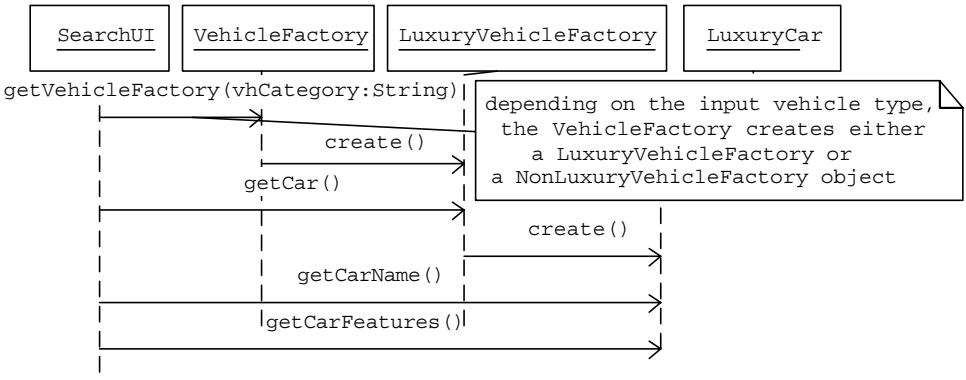

Figure 12.8 Luxury Car Details Query: Message Flow

EXAMPLE II

Let us design a simple customer data management application with the following features:

- The basic functionality is to validate and save the input customer data consisting of account, address and credit card data.
- The application should be functional in both local and remote modes.
- In the remote mode, the application should make use of remote objects using Java Remote Method Invocation (RMI) and save the data to a central server.

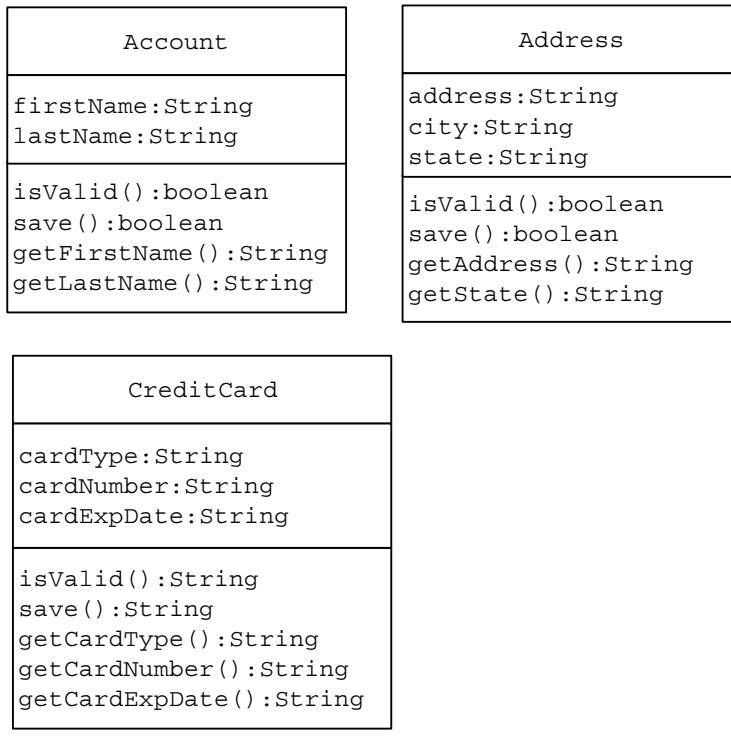

Figure 12.9 Classes Representing the Customer Data

- When the remote server is not available, users should be able to operate the application locally without interruption.
- The process of synchronizing both the local and the central databases is not considered as part of this example application.

Let us design three classes — `Account`, `Address` and `CreditCard` — as in Figure 12.9 representing the three different parts of the customer data.

Each of the customer data classes in Figure 12.9 offers methods to accept, validate and save appropriate parts of the customer data. Instances of these classes can be used by the application while operating in the local mode. As stated earlier, the application must function in remote mode as well and should make use of remote objects using RMI. The set of customer data classes in Figure 12.9 cannot be readily used as remote objects. This is because for a class instance to be accessible via RMI as a remote object, the class must:

- Extend the built-in `java.rmi.server.UnicastRemoteObject` class
- Implement the built-in `java.rmi.Remote` interface or any interface that is derived from the `java.rmi.Remote` interface
- Declare all of its methods to throw the built-in `java.rmi.RemoteException` exception

Hence, for the application to be functional in the remote mode, a second set of customer data classes is needed, whose instances can be accessed using RMI. In

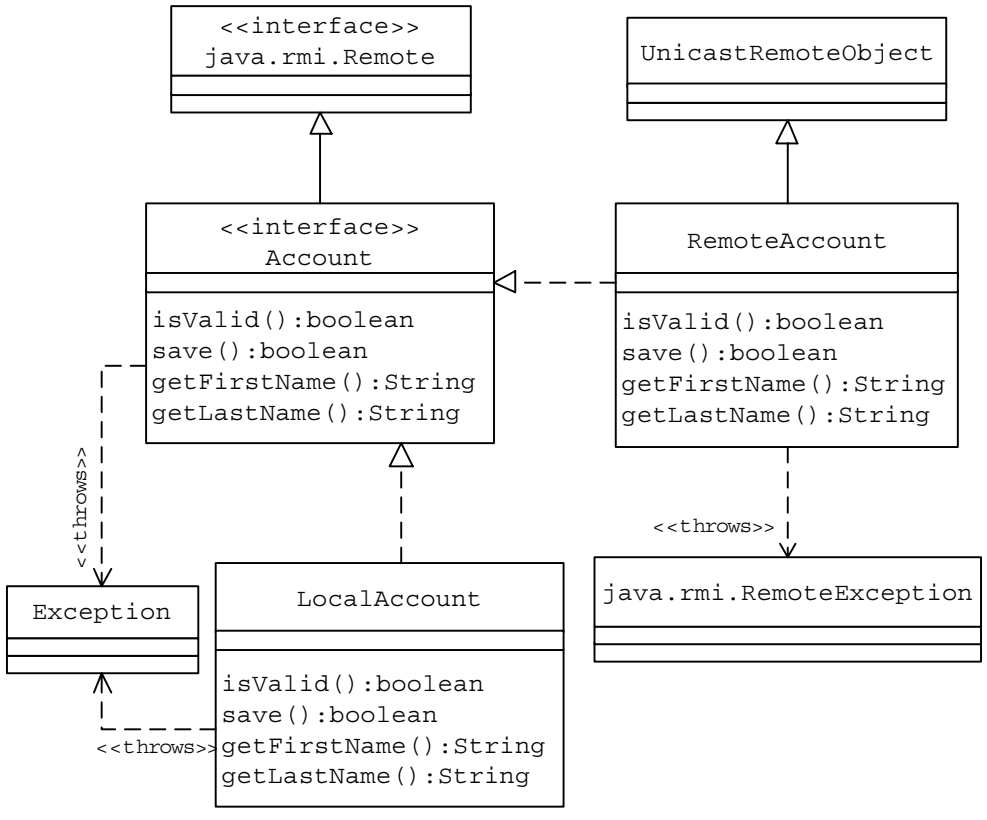

Figure 12.10 `Account` Class Hierarchy

addition, both the local and the remote objects must offer the same interface to enable the application to use both local and remote objects in a seamless manner.

The common interface for both the local and the remote class representations of the account data can be designed in the form of a Java interface `Account` (Figure 12.10) with the following features:

- Derived from the built-in `java.rmi.Remote` interface to enable its implementers to be used as remote objects accessible via RMI.
- Declares all of its methods to throw the `java.lang.Exception` exception. This allows an implementer class to throw any subclass of the `java.lang.Exception` class as part of implementing the interface methods.

As seen earlier, a remote class must declare all of its methods to throw the `java.rmi.RemoteException`. Because the `java.rmi.RemoteException` is a subclass of the `java.lang.Exception` class, the remote class representation of the account data can safely declare its methods to throw the `java.rmi.RemoteException` at the time of implementing the common `Account` interface methods. Because there are no special requirements for the local class representation of the customer account data, the local class can be designed to implement the same

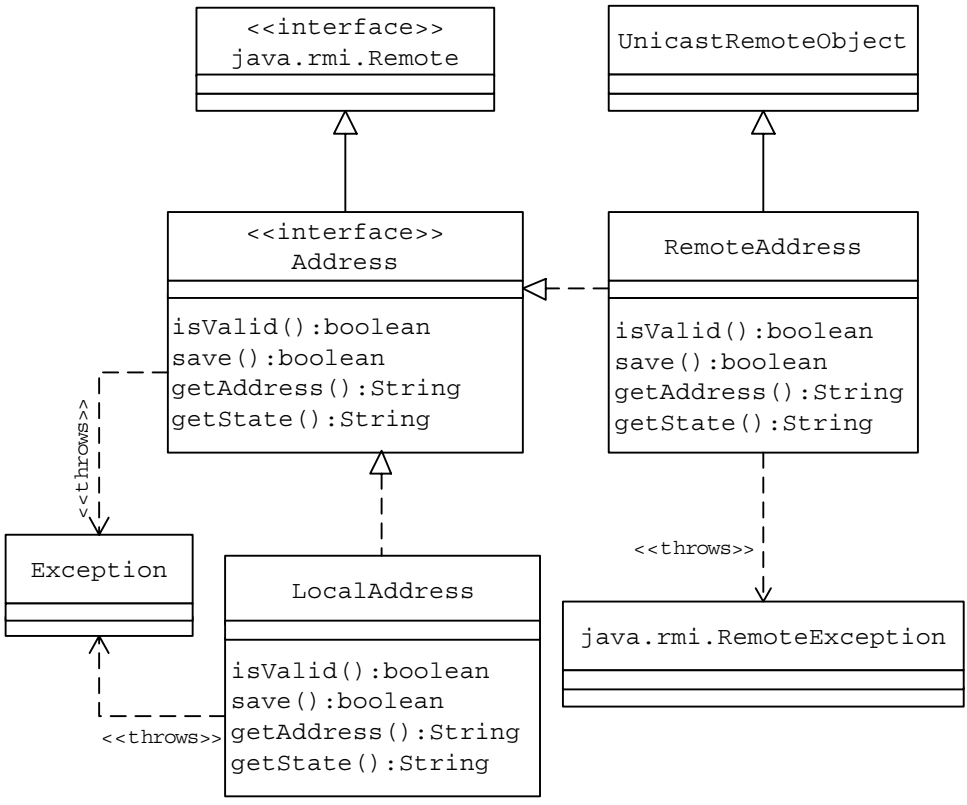

Figure 12.11 **`Address` Class Hierarchy**

`Account` interface as the remote class. As a result, both `LocalAccount` and `RemoteAccount` classes, which are the local and remote class representations of the account data, share the same common `Account` interface. Figure 12.10 shows the `Account` class hierarchy and its relationship with exception and RMI-specific classes.

In a similar manner, the class representations for both the address and the credit card parts of the customer data can be designed as in Figures 12.11 and 12.12, respectively.

Let us design an interface in the form of the `CustomerFactory` Java interface, which declares the methods to create instances of local or remote family of customer data classes (`Account`, `Address` and `CreditCard`). Let us further design two concrete implementers of the `CustomerFactory` interface as detailed in Table 12.3.

In this design, the `CustomerFactory` interface acts as an abstract factory and each of its concrete implementers act as factories. Figure 12.13 depicts the resulting factory class hierarchy.

To eliminate the need for a client object to deal with the factory objects directly, let us design a utility class `CustomerUtil` with a static method `getCustFactory(String mode)` that takes the current mode of operation as input and returns an appropriate `CustomerFactory` implementer object to the calling client object.

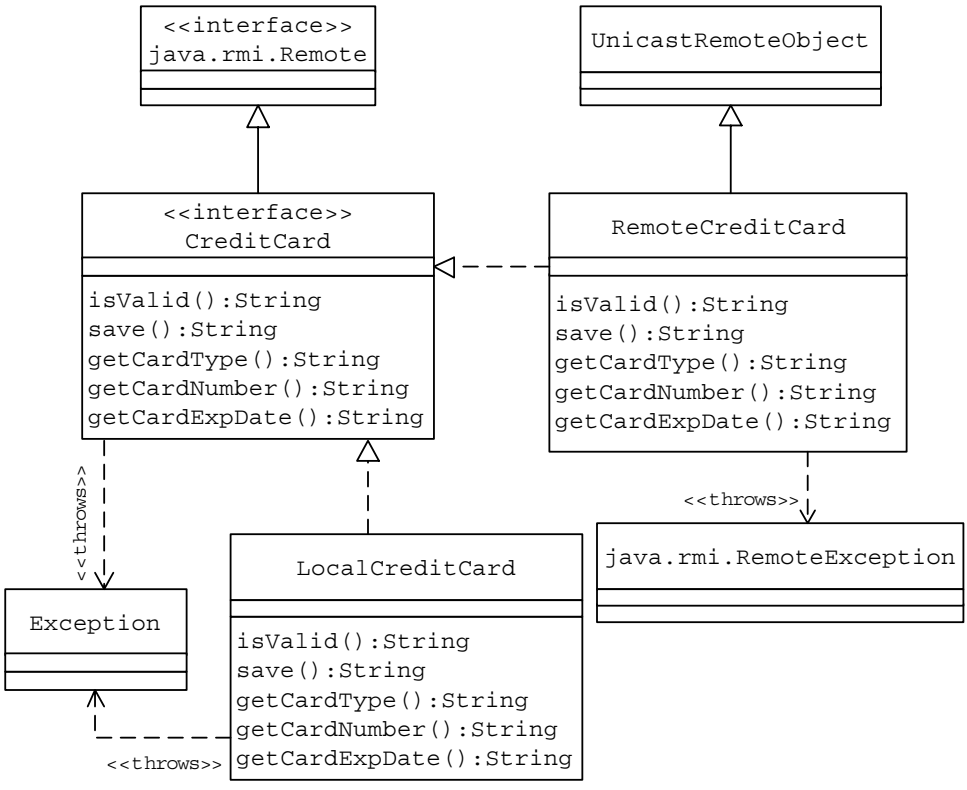

Figure 12.12 `CreditCard` Class Hierarchy

Table 12.3 Concrete Customer Factory Classes

Implementer Class	*Responsibility*
`LocalCustomerFactory`	Responsible for creating instances of classes representing the customer data in the local mode: `LocalAccount`, `LocalAddress` and `LocalCreditCard`
`RemoteCustomerFactory`	Responsible for creating instances of classes representing the customer data in the remote mode: `RemoteAccount`, `RemoteAddress` and `RemoteCreditCard`

Logical Flow When the Application Is Run

- The client object that needs to access the services of the customer data objects to validate and save the customer data is assumed to be aware of the current mode of operation (i.e., local or remote).

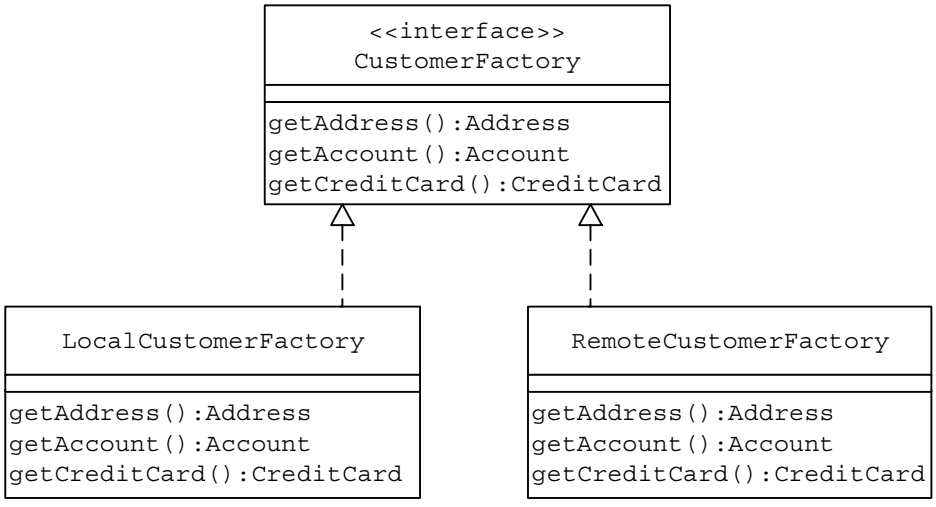

Figure 12.13 `CustomerFactory` Class Hierarchy

- By passing the application mode as the parameter, the client object can invoke the `getCustFactory(String)` static method of the `CustomerUtil` class.
- Inside the `getCustFactory(String)` method, the `CustomerUtil` creates an appropriate customer factory object and returns it to the client as an object of the `CustomerFactory` type.
- The client object can create the required customer data class (`Account`, `Address` or `CreditCard`) by invoking the `CustomerFactory` methods on the returned factory instance.
- Once an appropriate customer data object is created, the class object can use it to validate and save data.

PRACTICE QUESTIONS

1. Implement the example-II customer data management application.
2. Draw sequence diagrams to depict the message flow when the application is run.
3. Consider a Web hosting company that offers hosting services on both Windows and UNIX platforms. Suppose that the Web hosting company offers three different types of hosting packages — Basic, Premium and Premium Plus — on both platforms. Design an application using the Abstract Factory pattern to query the features of different types of hosting packages offered by the Web hosting company.