20

ADAPTER

This pattern was previously described in GoF95.

DESCRIPTION

In general, clients of a class access the services offered by the class through its interface. Sometimes, an existing class may provide the functionality required by a client, but its interface may not be what the client expects. This could happen due to various reasons such as the existing interface may be too detailed, or it may lack in detail, or the terminology used by the interface may be different from what the client is looking for.

In such cases, the existing interface needs to be converted into another interface, which the client expects, preserving the reusability of the existing class. Without such conversion, the client will not be able to use the functionality offered by the class. This can be accomplished by using the Adapter pattern. The Adapter pattern suggests defining a wrapper class around the object with the incompatible interface. This wrapper object is referred as an *adapter* and the object it wraps is referred to as an *adaptee*. The adapter provides the required interface expected by the client. The implementation of the adapter interface converts client requests into calls to the adaptee class interface. In other words, when a client calls an adapter method, internally the adapter class calls a method of the adaptee class, which the client has no knowledge of. This gives the client indirect access to the adaptee class. Thus, an adapter can be used to make classes work together that could not otherwise because of incompatible interfaces.

The term *interface* used in the discussion above:

- Does *not* refer to the concept of an interface in Java programming language, though a class's interface may be declared using a Java interface.
- Does *not* refer to the user interface of a typical GUI application consisting of windows and GUI controls.
- Does refer to the programming interface that a class exposes, which is meant to be used by other classes. As an example, when a class is designed as an abstract class or a Java interface, the set of methods declared in it makes up the class's interface.

CLASS ADAPTERS VERSUS OBJECT ADAPTERS

Adapters can be classified broadly into two categories — class adapters and object adapters — based on the way a given adapter is designed.

Class Adapter

A class adapter is designed by subclassing the adaptee class. In addition, a class adapter implements the interface expected by the client object. When a client object invokes a class adapter method, the adapter internally calls an adaptee method that it inherited.

Object Adapter

An object adapter contains a reference to an adaptee object. Similar to a class adapter, an object adapter also implements the interface, which the client expects. When a client object calls an object adapter method, the object adapter invokes an appropriate method on the adaptee instance whose reference it contains. Table 20.1 lists the differences between class and object adapters in detail.

EXAMPLE

Let us build an application to validate a given customer address. This application can be part of a larger customer data management application.

Let us define a `Customer` class as in Figure 20.1 (Listing 20.1).

Different client objects can create a `Customer` object and invoke the `isValidAddress` method to check the validity of the customer address. For the purpose of validating the address, the `Customer` class expects to make use of an address validator class that provides the interface declared in the `AddressValidator` interface (Listing 20.2).

Let us define one such validator `USAddress` to validate a given U.S. address as in Listing 20.3.

The `USAddress` class is designed to implement the `AddressValidator` interface so that `Customer` objects can use `USAddress` instances as part of the customer address validation process without any problems (Listing 20.4). Figure 20.2 shows the class association.

Let us say that the application needs to be enhanced to deal with customers from Canada as well. This requires a validator for verifying the addresses of Canadian customers. Let us assume that a utility class `CAAddress`, with the required functionality to validate a given Canadian address, already exists.

From the `CAAddress` class implementation in Listing 20.5, it can be observed that the `CAAddress` does offer the validation service required by the `Customer` class, but the interface it offers is different from what the `Customer` class expects.

The `CAAddress` class offers an `isValidCanadianAddr` method, but the `Customer` expects an `isValidAddress` method as declared in the `AddressValidator` interface.

This incompatibility in the interface makes it difficult for a `Customer` object to use the existing `CAAddress` class. One of the options is to change the interface

Table 20.1 Class Adapters versus Object Adapters

Class Adapters	*Object Adapters*
Based on the concept of inheritance.	Uses object composition.
Can be used to adapt the interface of the adaptee only. Cannot adapt the interfaces of its subclasses, as the adapter is statically linked with the adaptee when it is created.	Can be used to adapt the interface of the adaptee and all of its subclasses.
Because the adapter is designed as a subclass of the adaptee, it is possible to override some of the adaptee's behavior. **Note:** In Java, a subclass cannot override a method that is declared as final in its parent class.	Cannot override adaptee methods. **Note:** Literally, cannot "override" simply because there is no inheritance. But wrapper functions provided by the adapter can change the behavior as required.
The client will have some knowledge of the adatee's interface as the full public interface of the adaptee is visible to the client.	The client and the adaptee are completely decoupled. Only the adapter is aware of the adaptee's interface.
In Java applications: Suitable when the expected interface is available in the form of a Java interface and not as an abstract or concrete class. This is because the Java programming language allows only single inheritance. Since a class adapter is designed as a subclass of the adaptee class, it will not be able to subclass the interface class (representing the expected interface) also, if the expected interface is available in the form of an abstract or concrete class.	In Java applications: Suitable even when the interface that a client object expects is available in the form of an abstract class. Can also be used if the expected interface is available in the form of a Java interface. Or When there is a need to adapt the interface of the adaptee and also all of its subclasses.
In Java applications: Can adapt methods with protected access specifier.	In Java applications: Cannot adapt methods with protected access specifier, unless the adapter and the adaptee are designed to be part of the same package.

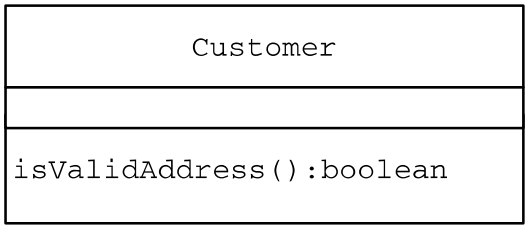

Figure 20.1 `Customer` Class

Listing 20.1 `Customer` Class

```
class Customer {
  public static final String US = "US";
  public static final String CANADA = "Canada";
  private String address;
  private String name;
  private String zip, state, type;
  public boolean isValidAddress() {
          …
          …
  }
  public Customer(String inp_name, String inp_address,
                  String inp_zip, String inp_state,
                  String inp_type) {
    name = inp_name;
    address = inp_address;
    zip = inp_zip;
    state = inp_state;
    type = inp_type;
  }
}//end of class
```

Listing 20.2 `AddressValidator` as an Interface

```
public interface AddressValidator {
  public boolean isValidAddress(String inp_address,
     String inp_zip, String inp_state);
}//end of class
```

of the `CAAddress` class, but it is not advisable as there could be other applications using the `CAAddress` class in its current form. Changing the `CAAddress` class interface can affect all of those current clients of the `CAAddress` class.

Applying the Adapter pattern, a class adapter `CAAddressAdapter` can be designed as a subclass of the `CAAddress` class implementing the `AddressValidator` interface (Figure 20.3 and Listing 20.6).

Because the adapter `CAAddressAdapter` implements the `AddressValidator` interface, client objects can access the adapter `CAAddressAdapter` objects without any problems. When a client object invokes the `isValidAddress` method

Listing 20.3 `USAddress` Class

```
class USAddress implements AddressValidator {
  public boolean isValidAddress(String inp_address,
      String inp_zip, String inp_state) {
    if (inp_address.trim().length() < 10)
      return false;
    if (inp_zip.trim().length() < 5)
      return false;
    if (inp_zip.trim().length() > 10)
      return false;
    if (inp_state.trim().length() != 2)
      return false;
    return true;
  }
}//end of class
```

Listing 20.4 `Customer` Class Using the `USAddress` Class

```
class Customer {
          …
          …
  public boolean isValidAddress() {
    //get an appropriate address validator
    AddressValidator validator = getValidator(type);
    //Polymorphic call to validate the address
    return validator.isValidAddress(address, zip, state);
  }
  private AddressValidator getValidator(String custType) {
    AddressValidator validator = null;
    if (custType.equals(Customer.US)) {
      validator = new USAddress();
    }
    return validator;
  }
}//end of class
```

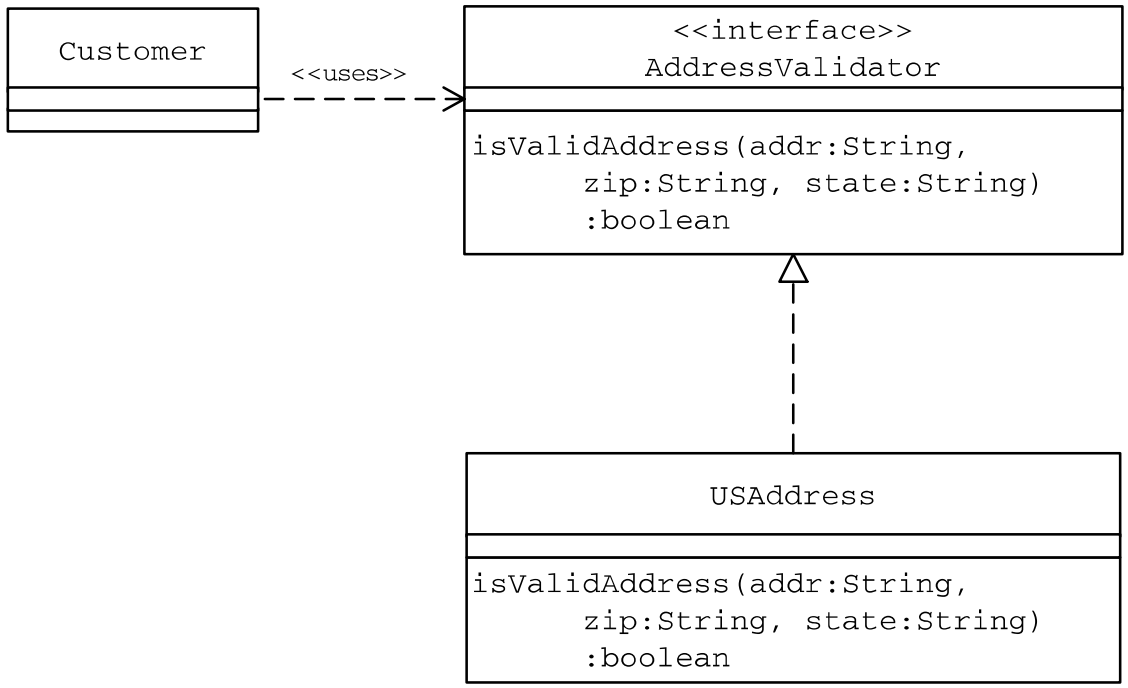

Figure 20.2 **`Customer/USAddress Validator`: Class Association**

Listing 20.5 **`CAAdress` Class with Incompatible Interface**

```
class CAAddress {
  public boolean isValidCanadianAddr(String inp_address,
     String inp_pcode, String inp_prvnc) {
    if (inp_address.trim().length() < 15)
      return false;
    if (inp_pcode.trim().length() != 6)
      return false;
    if (inp_prvnc.trim().length() < 6)
      return false;
    return true;
  }
}//end of class
```

on the adapter instance, the adapter internally translates it into a call to the inherited `isValidCanadianAddr` method.

Inside the `Customer` class, the `getValidator` private method needs to be enhanced so that it returns an instance of the `CAAddressAdapter` in the case of Canadian customers (Listing 20.7). The polymorphic call on the returned object (inside the `isValidAddress` method) does not need to be changed as both the `USAddress` and `CAAddressAdapter` implement the same `AddressValidator` interface.

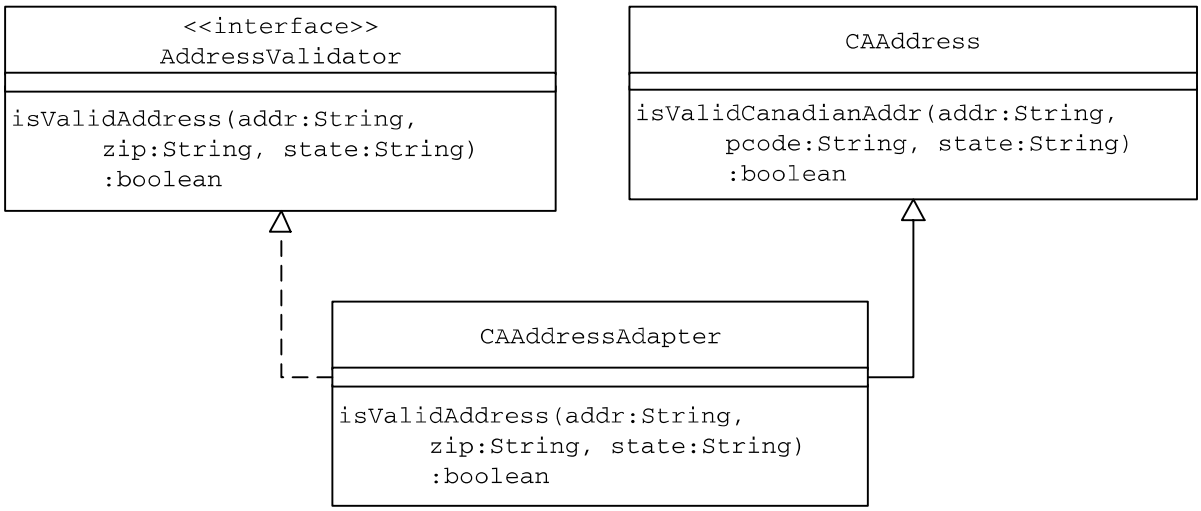

Figure 20.3 Class Adapter for the `CAAddress` Class

Listing 20.6 `CAAddressAdapter` as a Class Adapter

```
public class CAAddressAdapter extends CAAddress
  implements AddressValidator {
  public boolean isValidAddress(String inp_address,
     String inp_zip, String inp_state) {
    return isValidCanadianAddr(inp_address, inp_zip,
           inp_state);
  }
}//end of class
```

The combination of the `CAAddressAdapter` design and the polymorphic call on an object of the `AddressValidator` (that declares the expected interface) type object enables the `Customer` to make use of the services of the `CAAddress` class that has an incompatible interface.

The class diagram in Figure 20.4 shows the overall class association.

The sequence diagram in Figure 20.5 depicts the message flow when the `CAAddressAdapter` is designed as a class adapter.

ADDRESS ADAPTER AS AN OBJECT ADAPTER

While discussing the design of the address adapter as a class adapter, we saw that the `AddressValidator` interface expected by the client is defined in the form of a Java interface. Now let us assume that the client expects the `AddressValidator` interface to be available as an abstract class instead of a Java interface (Listing 20.8). Because the adapter `CAAdapter` has to provide the interface declared by the `AddressValidator` abstract class, the adapter needs to be

Listing 20.7 `Customer` Class Using the `CAAddressAdapter` Class

```
class Customer {
          …
          …
  public boolean isValidAddress() {
    //get an appropriate address validator
    AddressValidator validator = getValidator(type);
    //Polymorphic call to validate the address
    return validator.isValidAddress(address, zip, state);
  }
  private AddressValidator getValidator(String custType) {
    AddressValidator validator = null;
    if (custType.equals(Customer.US)) {
      validator = new USAddress();
    }
    if (type.equals(Customer.CANADA)) {
      validator = new CAAddressAdapter();
    }
    return validator;
  }
}//end of class
```

designed to subclass the `AddressValidator` abstract class and implement its abstract methods (Listing 20.9).

Because multiple inheritance is not supported in Java, now the adapter `CAAddressAdapter` cannot subclass the existing `CAAddress` class as it has already used its only chance to subclass from another class.

Applying the *object* Adapter pattern, the `CAAddressAdapter` can be designed to contain an instance of the adaptee `CAAddress` (Figure 20.6 and Listing 20.10). This adaptee instance is passed to the adapter by its clients, when the adapter is first created. In general, the adaptee instance contained by an object adapter may be provided in the following two ways:

- Clients of the object adapter may pass the adaptee instance to the adapter. This approach is more flexible in choosing the class to adapt from, but then the client may become aware of the adaptee or the fact of adaptation. It is more suitable when the adapter needs any specific state from the adaptee object besides its behavior.
- The adapter may create the adaptee instance on its own. This approach is relatively less flexible and suitable when the adapter does not need any specific state from the adaptee, but needs only its behavior.

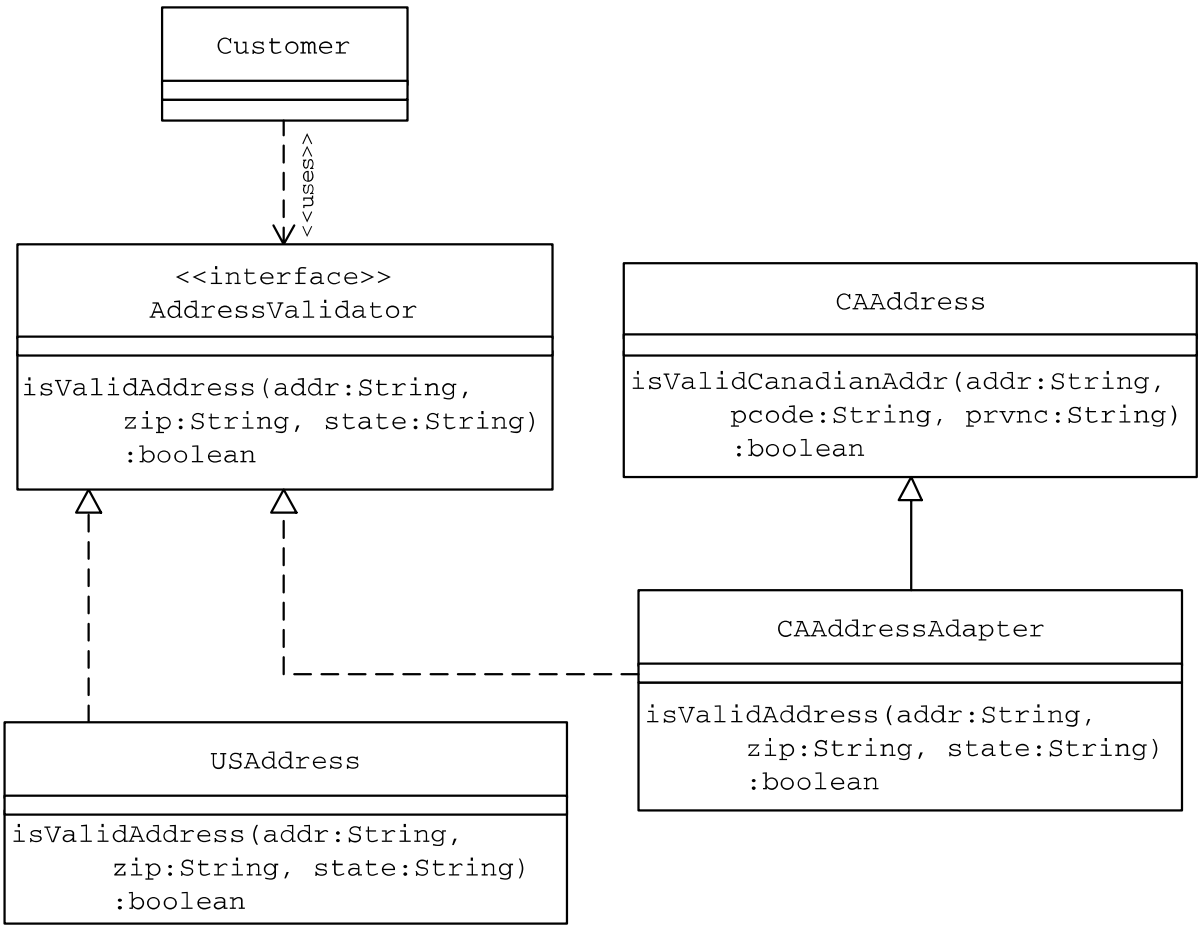

Figure 20.4 Address Validation Application: Using Class Adapter

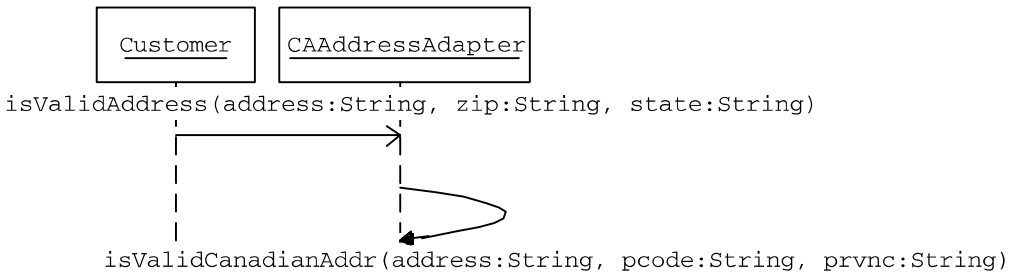

Figure 20.5 Address Validation Message Flow: Using Class Adapter

Listing 20.8 `AddressValidator` as an Abstract Class

```
public abstract class AddressValidator {
  public abstract boolean isValidAddress(String inp_address,
     String inp_zip, String inp_state);
}//end of class
```

Listing 20.9 `CAAddressAdapter` Class

```
class CAAddressAdapter extends AddressValidator {
          …
          …
  public CAAddressAdapter(CAAddress address) {
    objCAAddress = address;
  }
  public boolean isValidAddress(String inp_address,
     String inp_zip, String inp_state) {
          …
          …
  }
}//end of class
```

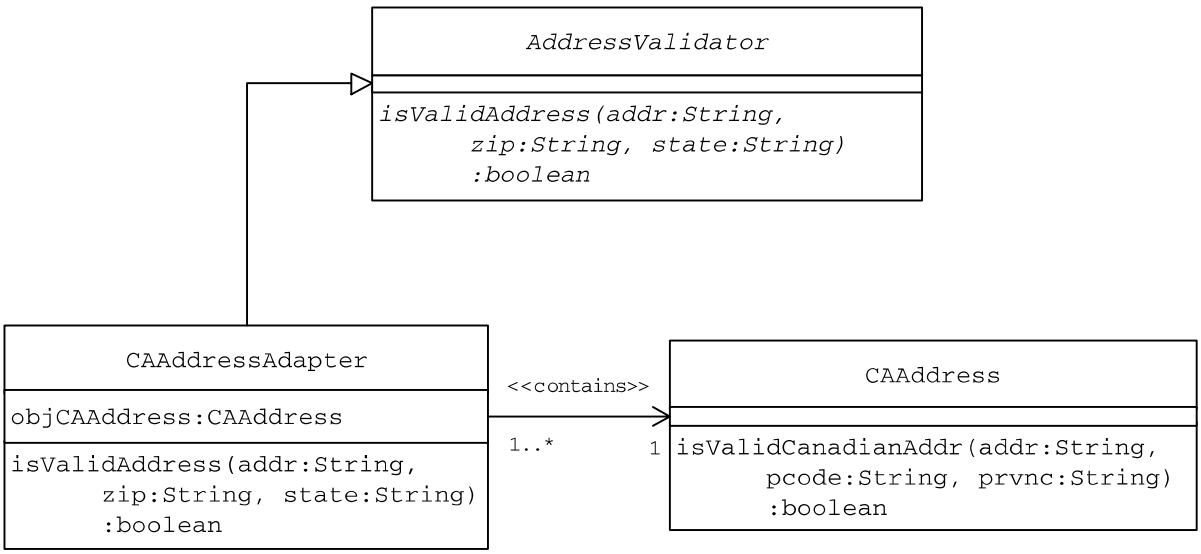

Figure 20.6 Object Adapter for the `CAAddress` Class

When a client object invokes the `isValidAddress` method on a `CAAddressAdapter` (adapter) instance, the adapter internally calls the `isValidCanadianAddr` method on the `CAAddress` (adaptee) instance it contains.

The class diagram in Figure 20.7 shows the overall class association when the address adapter is designed as an object adapter.

From the example application design it can be observed that an adapter enables the `Customer` (client) class to access the services offered by the `CAAddress` (adaptee) with an incompatible interface.

The sequence diagram in Figure 20.8 shows the message flow when the adapter `CAAddressAdapter` is designed as an object adapter.

Listing 20.10 `CAAddressAdapter` as an Object Adapter

```
class CAAddressAdapter extends AddressValidator {
  private CAAddress objCAAddress;
  public CAAddressAdapter(CAAddress address) {
    objCAAddress = address;
  }
  public boolean isValidAddress(String inp_address,
      String inp_zip, String inp_state) {
    return objCAAddress.isValidCanadianAddr(inp_address,
           inp_zip, inp_state);
  }
}//end of class
```

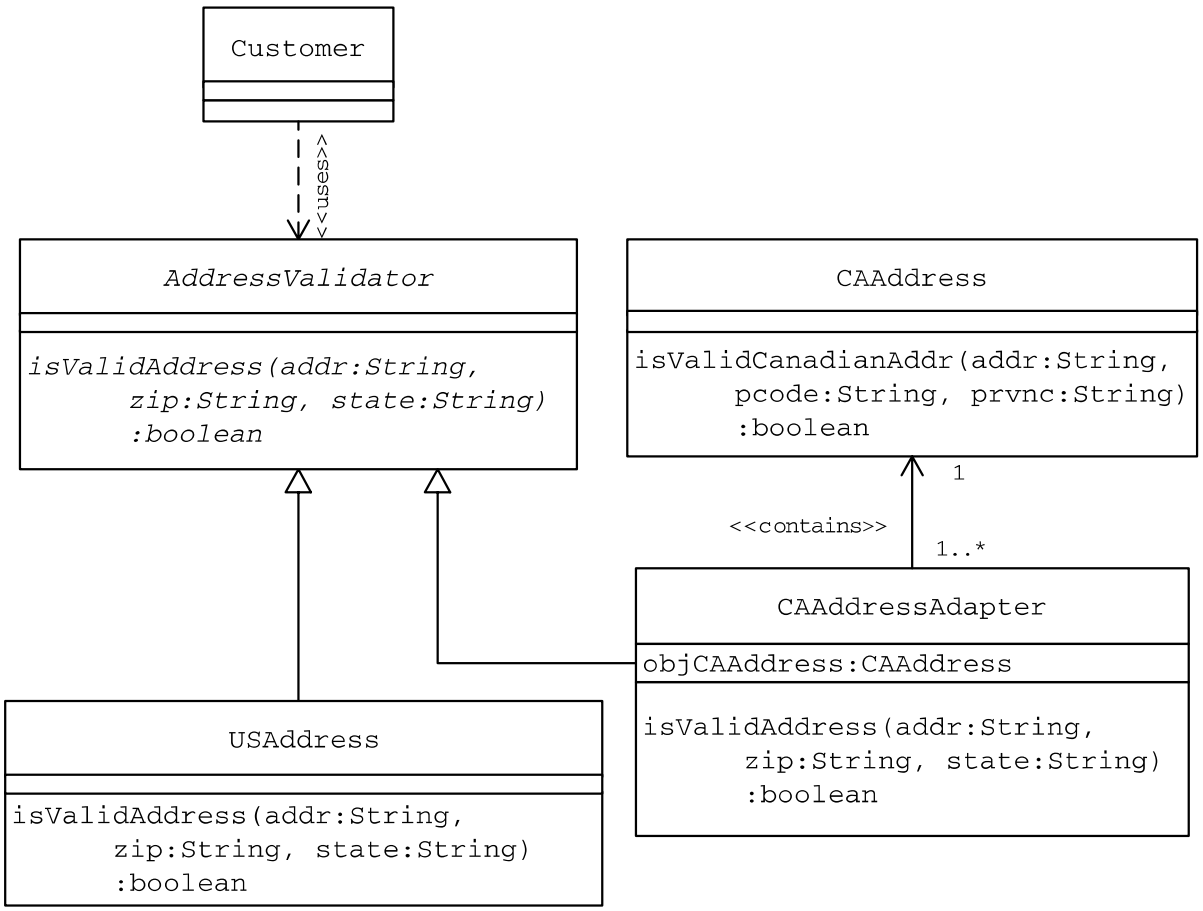

Figure 20.7 Address Validation Application: Using Object Adapter

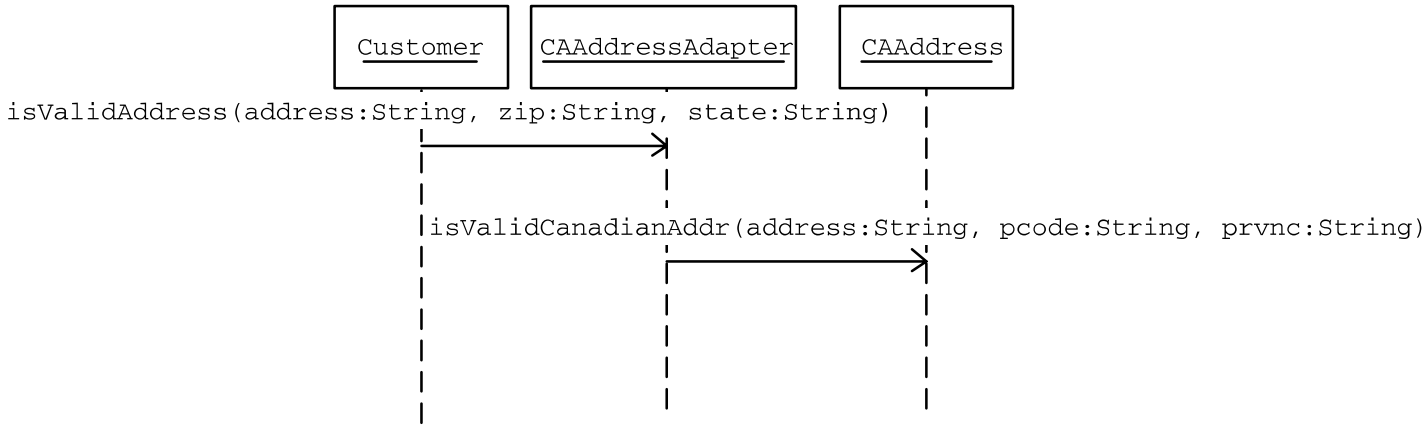

Figure 20.8 Address Validation Message Flow: Using Object Adapter

PRACTICE QUESTIONS

1. During the discussion of the Factory Method pattern, we designed a message logging class `FileLogger` with a method `log(String)` that can be used by client objects to log messages (Figure 20.9). Let us assume that a client `LoggerClient` expects a message logging class to provide an interface as follows:

```
public abstract class LoggerIntr{
  public abstract boolean logMessage(String msg);
}//end of class
```

   How would you design an adapter, say `FileLoggerAdapter`, to adapt the `FileLogger` class's existing interface?
2. In the above practice question, if the client `LoggerClient` expects a message logging class to provide an interface as follows:

```
public interface LoggerIntr{
  public boolean logMessage(String msg);
}//end of interface
```

   Can you design the adapter `FileLoggerAdapter` as a class adapter, an object adapter or both?
3. Design two subclasses — `HTMLFileLogger` and `EncFileLogger` — of the `FileLogger` as in Table 20.2. Each of these subclasses override the `log(String)` method of the parent `FileLogger` class to provide the required functionality. Assume that the client `LoggerClient` requires the functionality offered by the `FileLogger` and also its subclasses. If the client `LoggerClient` expects the message logging class to provide an interface as follows:

```
public interface LoggerIntr{
  public boolean logMessage(String msg);
}//end of interface
```

   Can you design the adapter `FileLoggerAdapter` as a class adapter, an object adapter or both? Why?

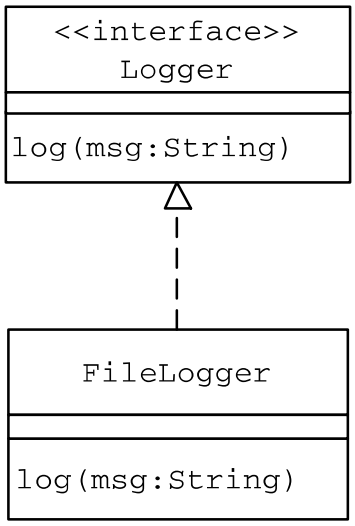

Figure 20.9 Messaging Logging Utility

Table 20.2 `FileLogger` Subclasses

Subclass	*Functionality*
`HTMLFileLogger`	Transform an incoming message into an HTML document and store it in a log file.
`EncFileLogger`	Apply encryption on an incoming message and store it in a log file.