INSTRUCTION:
Design and implement a Java program for the following description of an *Address Validation* application. You must apply *Adapter* Design Pattern.

An application is built to validate a given customer address. This application can be part of a larger customer data management application.

A `Customer` class is defined as below:

```
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
```

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.

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

One such validator `USAddress` to validate a given U.S. address is defined as below:

```
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
```

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 as shown below:

```
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
```

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 below, 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.

```
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
```

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 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.

1. Solve the incompatible interface problem above by using *Class Adapter* pattern. Draw a UML class diagram to show your design for the *Address Validation* application.

2. Implement the *Address Validation* application in Java based on your design above.

3. Create a test class (AddressClassAdapterTest.java) to test your implementation. The output should be as shown below:

```
Customer Name: Google
Address: 1600 Amphitheatre Parkway
Zip/PostalCode: 94043
State/Province: CA
Address Type: US
Result: Valid customer data

Customer Name: Google
Address: 1600 Amphitheatre Parkway
Zip/PostalCode: 94043
State/Province: CA
Address Type: Canada
Result: Invalid customer data
```

4. 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. Because the adapter `CAAddressAdapter` has to provide the interface declared by the `AddressValidator` abstract class, the adapter needs to be designed to subclass the `AddressValidator` abstract class and implement its abstract methods.

   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.

   Solve this problem by applying the *Object Adapter* pattern instead of *Class Adapter* pattern. Draw a UML class diagram to show your design.

5. Implement the *Address Validation* application in Java based on your new design above.

6. What changes do you have to make to your test driver class (AddressClassAdapterTest.java) to produce the same output as in No. 3 above?