Wrappers;

Decorator and Adapter Patterns

Mohsen Afsharchi

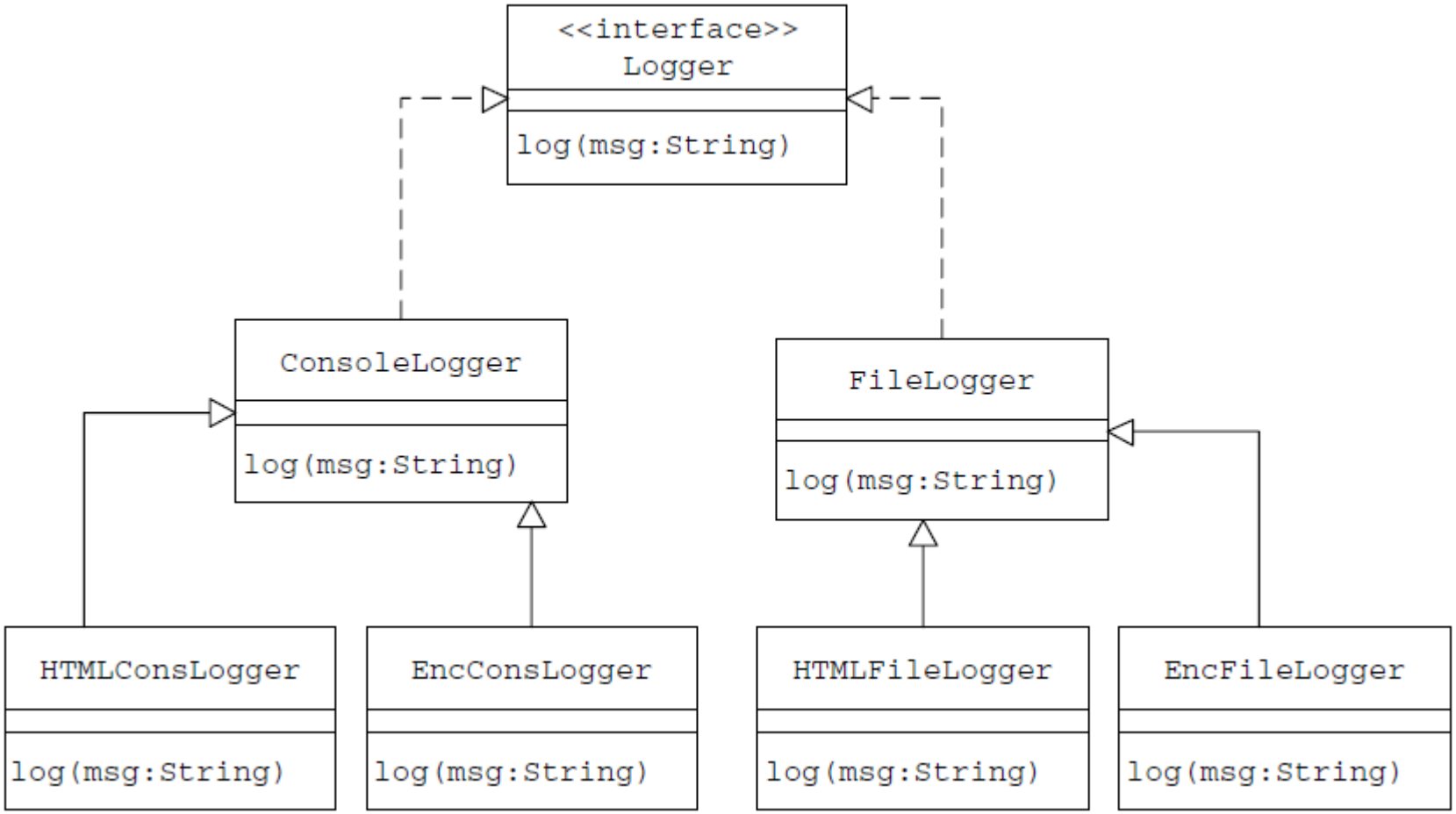
Decorator

- The `Decorator` object is designed to have the same interface as the underlying object. This allows a client object to interact with the `Decorator` object in exactly the same manner as it would with the underlying actual object.
- The `Decorator` object contains a reference to the actual object.
- The `Decorator` object receives all requests (calls) from a client. It in turn forwards these calls to the underlying object.
- The `Decorator` object adds some additional functionality before or after forwarding requests to the underlying object. This ensures that the additional functionality can be added to a given object externally at runtime without modifying its structure.

Decorator vs Inheritance

Decorator Pattern	*Inheritance*
Used to extend the functionality of a particular object.	Used to extend the functionality of a class of objects.
Does not require subclassing.	Requires subclassing.
Dynamic.	Static.
Runtime assignment of responsibilities.	Compile time assignment of responsibilities.
Prevents the proliferation of subclasses leading to less complexity and confusion.	Could lead to numerous subclasses, exploding class hierarchy on specific occasions.
More flexible.	Less flexible.
Possible to have different decorator objects for a given object simultaneously. A client can choose what capabilities it wants by sending messages to an appropriate decorator.	Having subclasses for all possible combinations of additional capabilities, which clients expect out of a given class, could lead to a proliferation of subclasses.
Easy to add any combination of capabilities. The same capability can even be added twice.	Difficult.

Example


<<interface>>
Logger
log(msg:String)
ConsoleLogger
log(msg:String)
FileLogger
log(msg:String)
HTMLConsLogger
log(msg:String)
EncConsLogger
log(msg:String)
HTMLFileLogger
log(msg:String)
EncFileLogger
log(msg:String)

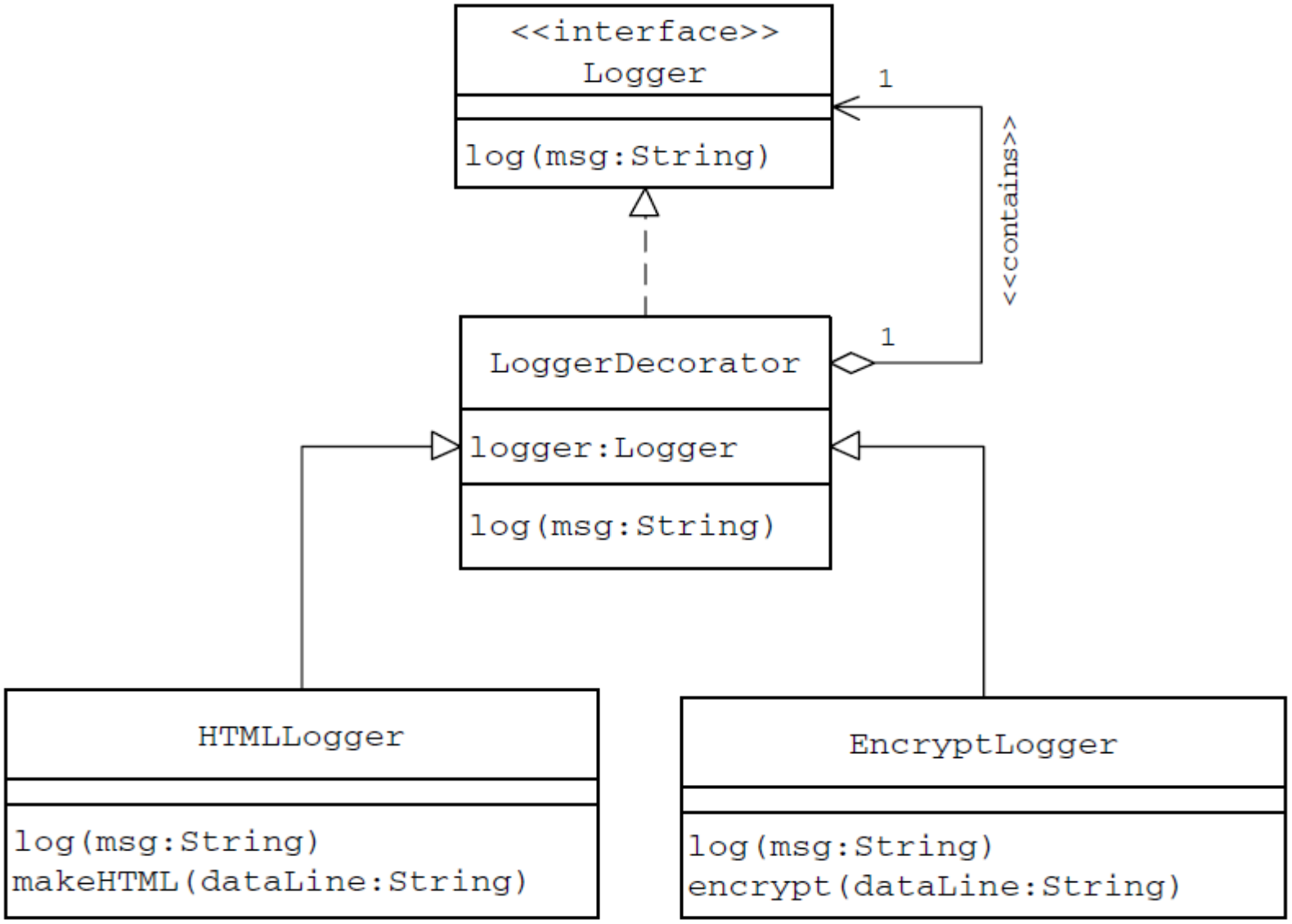
Example

Subclass	*Parent Class*	*Functionality*
`HTMLFileLogger`	`FileLogger`	Transform an incoming message to an HTML document and store it in a log file.
`HTMLConsLogger`	`ConsoleLogger`	Transform an incoming message to an HTML document and display it on the screen.
`EncFileLogger`	`FileLogger`	Apply encryption on an incoming message and store it in a log file.
`EncConsLogger`	`ConsoleLogger`	Apply encryption on an incoming message and display it on the screen.

LoggerDecorator Class

```
public class LoggerDecorator implements Logger {
  Logger logger;
  public LoggerDecorator(Logger inp_logger) {
    logger = inp_logger;
  }
  public void log(String DataLine) {
    /*
      Default implementation
      to be overriden by subclasses.
    */
    logger.log(DataLine);
  }
}//end of class
```

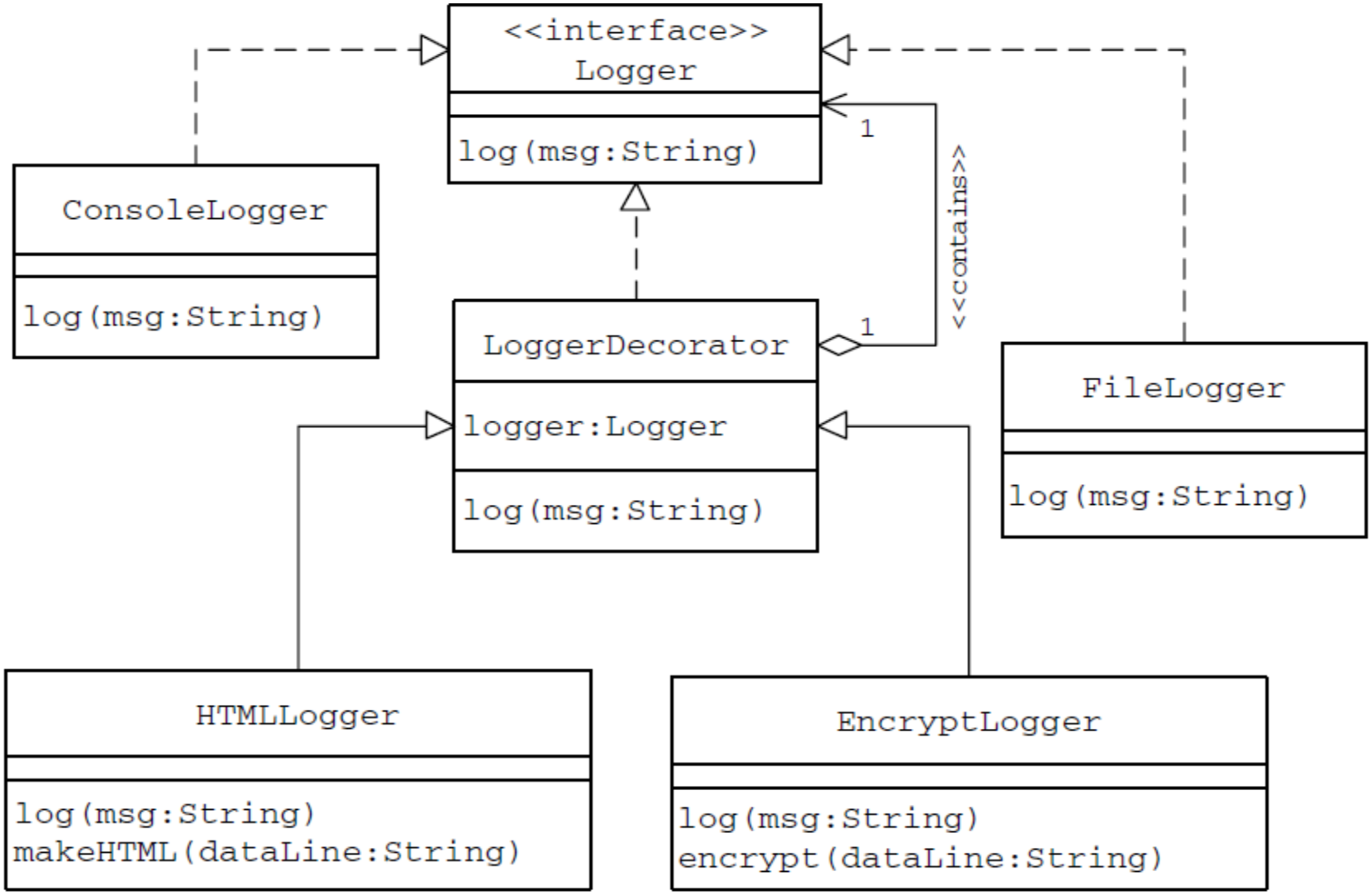
Class Structure



Class Code

```
public class HTMLLogger extends LoggerDecorator {
  public HTMLLogger(Logger inp_logger) {
    super(inp_logger);
  }
  public void log(String DataLine) {
    DataLine = makeHTML(DataLine);
    logger.log(DataLine);
  }
  public String makeHTML(String DataLine) {
    DataLine = "<HTML><BODY>" + "<b>" + DataLine +
      "</b>" + "</BODY></HTML>";
    return DataLine;
  }
}//end of class
```

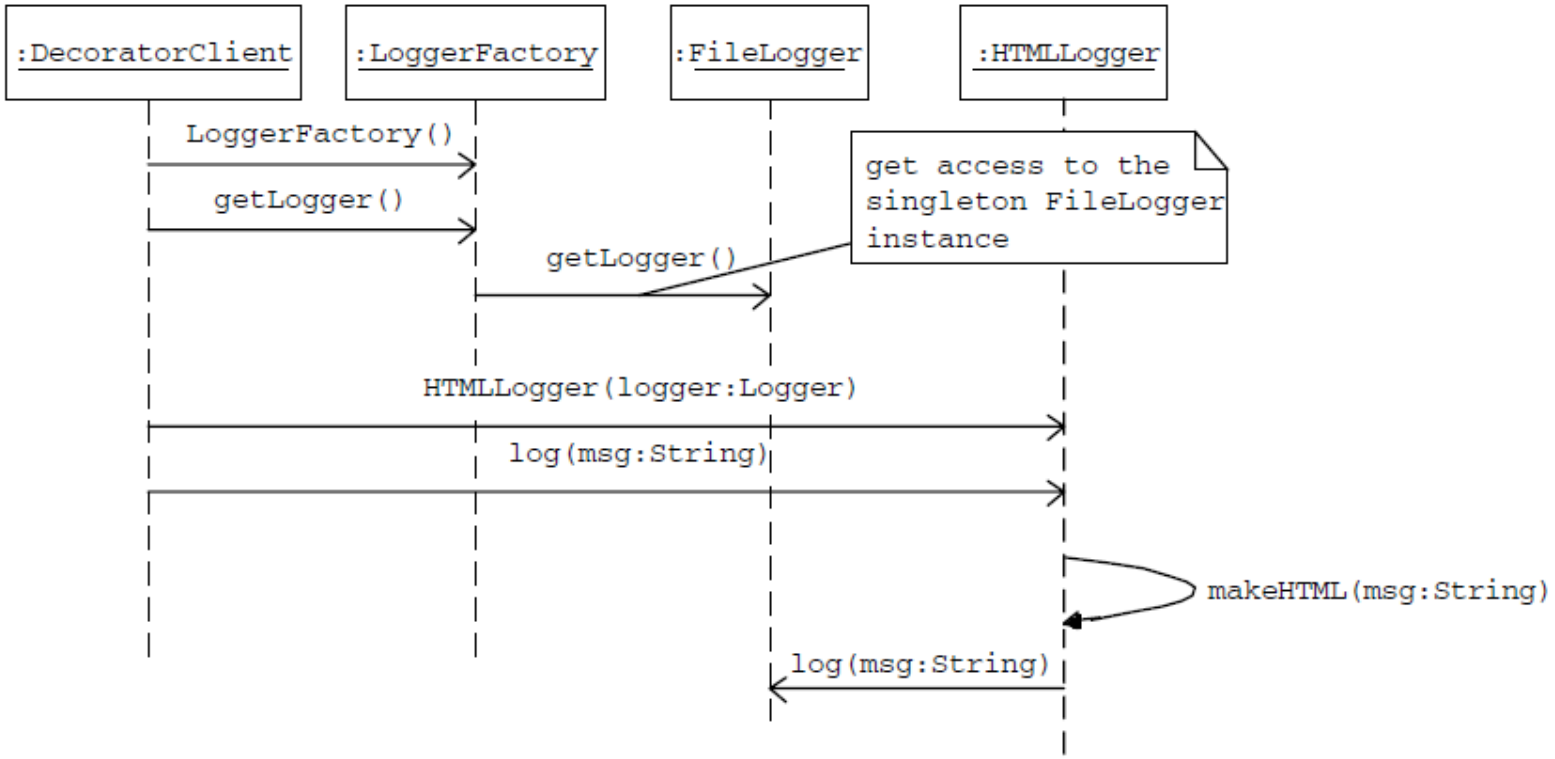
Class Hierarchy



Client Class

```
class DecoratorClient {
  public static void main(String[] args) {
    LoggerFactory factory = new LoggerFactory();
    Logger logger = factory.getLogger();
    HTMLLogger hLogger = new HTMLLogger(logger);
    //the decorator object provides the same interface.
    hLogger.log("A Message to Log");
    EncryptLogger eLogger = new EncryptLogger(logger);
    eLogger.log("A Message to Log");
  }
}//End of class
```

Message flow



Adapter

Adapter

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.

Class vs 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.

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
```

AddressValidator Class

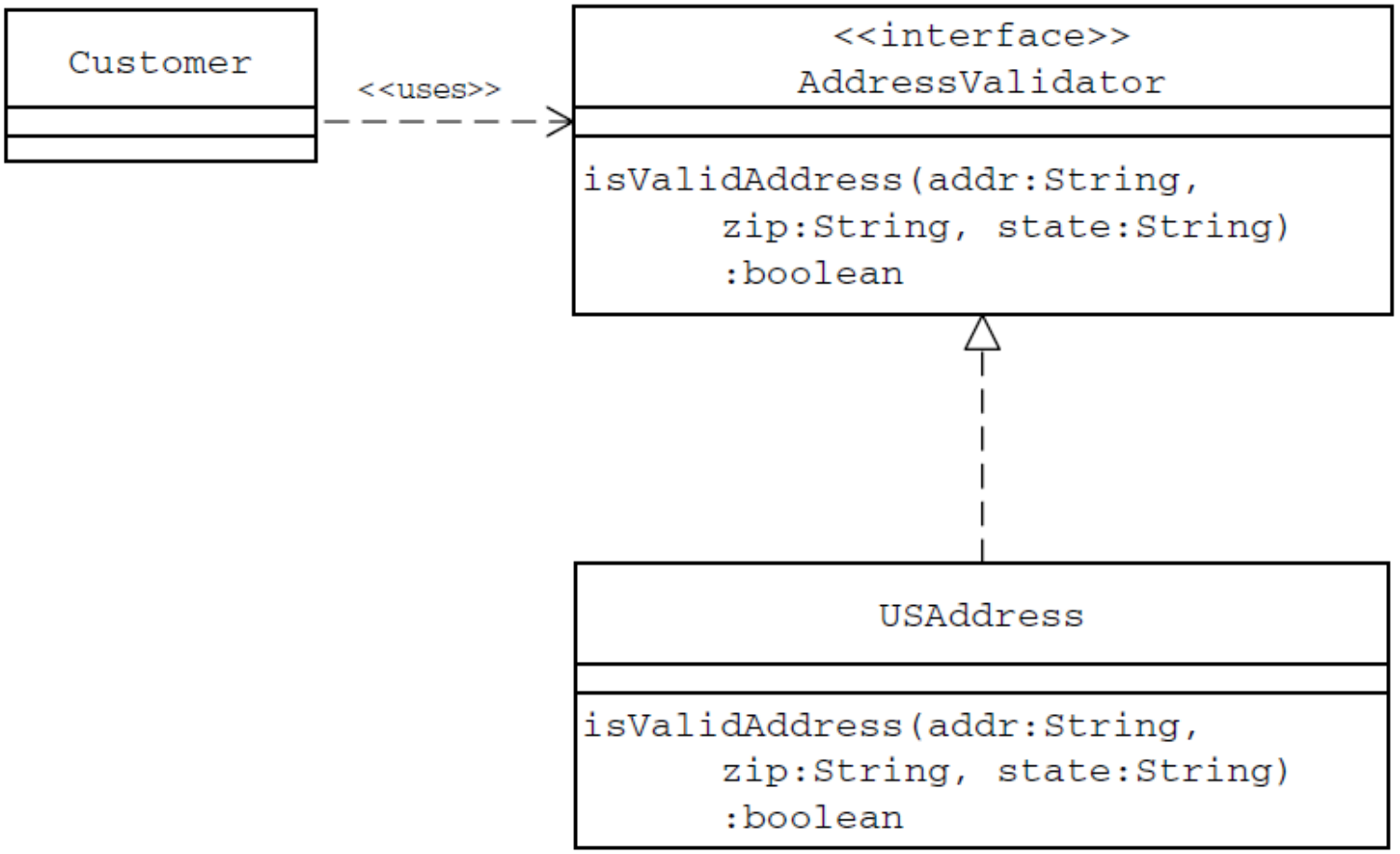
```
public interface AddressValidator {
  public boolean isValidAddress(String inp_address,
      String inp_zip, String inp_state);
}//end of class

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
```

Customer

```
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
```

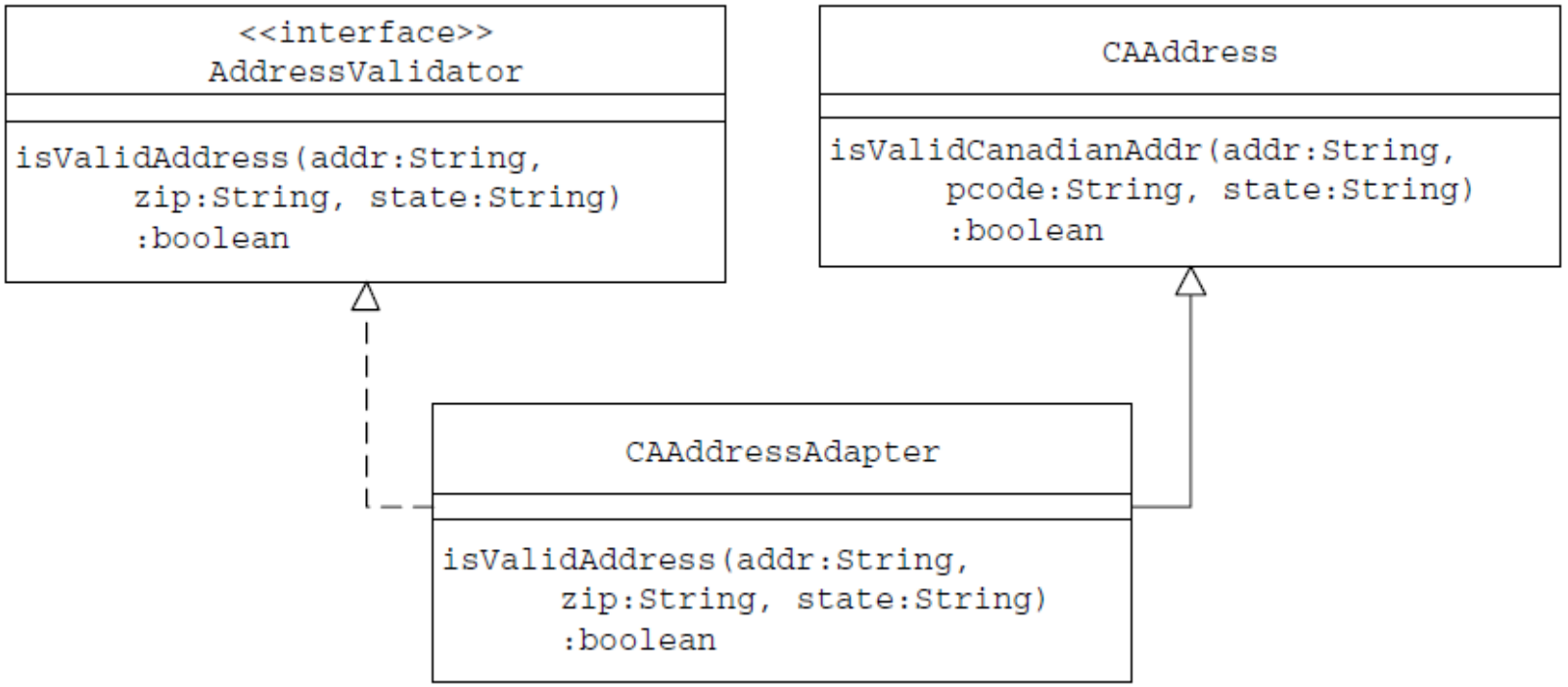
Class Association



Another Type of Address

```
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
```

Class Association



Class Extension

```
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
```

Validator

```
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;
}
```

Class Association