Validation Strategy

Matthew A. Brown
Senior Web Developer
Avery Dennison Corporation

Introduction

The Validation Strategy pattern concerns validation of an object's state with a focus on flexibility of implementation, and modularity of the object's behavior.

Context

In today's development environments, validating an object is frequently a requirement, regardless of software platform or language. For example, at our company, we perform validations on Data Transfer Objects (DTO) [1], XML objects, and even files objects. Let's use the DTO as an example. First, the DTO pattern uses a lightweight object to transfer data between the Enterprise Java Beans (EJB) layer and the client layer, allowing for greater performance. During our experience with DTOs, we found that we could use DTOs for more than simple web interfaces, but also for interfacing to legacy systems, file upload programs, and Java Messaging Services. Each one can be viewed as a different type of client. In all of these different contexts, we may or may not require validation of the object's state.

There are various types of validation that are used in environments today. Here is a listing of some of the types:

Validation Type	**Definition**	**Example of use**
Pattern Matching	Validation based on matching a pattern defined by a regular expression against the data entered. This type of validation allows you to check for predictable sequences of characters, such as those in social security numbers, e-mail addresses, telephone numbers, postal codes, and so on.	Validate email address for "@" and ".", with ending of net,com,org, ect. Regular Expressions typically use pattern matching.
Range Checking	Checks that a user's entry is between specified lower and upper boundaries. You can check ranges within pairs of numbers, alphabetic characters, and dates. Boundaries can be expressed as constants or as values derived from another control.	We check the future date of availability to make sure that the date is not before today.

Validation Type	Definition	Example of use
Comparison Checking	Compares a user's entry against a constant value, a value derived from another control, or a database value using a comparison operator (less than, equal, greater than, and so on).	We check the value of certain fields against a list of valid values defined in a properties file.
Cross Checking	Compares a field's value against other values entered, or values in databases/other systems/datasources for only a valid combination of values overall.	We check in the database for existing inventory "template" that provides default information. The template must exist first.
Type Checking	Checks to ensure that the field's value is of a specific type, such as an integer, String, character, ect.	We check to make sure that quantity, and numeric fields are only digit characters.
Null Checking	Checks to ensure that there is a value entered. Cannot be null.	We validate that the value is not null on required fields.

Combined with many differing contexts of the object's use, we need to think about code reuse. Fully utilizing an object's code base can be accomplished through allowing for flexibility in the design of the object's responsibilities.

Problem

How do you automate validation of the state of an object's fields that allows flexibility for a variety of implementations of the object?

Forces

In certain environments, modularity of code is paramount. With proper design, there should be clean separation of the responsibilities for objects (i.e. not mixing in business logic into the persistence objects). Normally, the state of an object, and the validation of that state, should be encapsulated within that object. However, we found that in differing contexts of using an object, we did not always require validation of the object's state in the same way. We wanted to maximize code reuse. Accordingly, avoiding implementation specific logic within DTO object is desired.

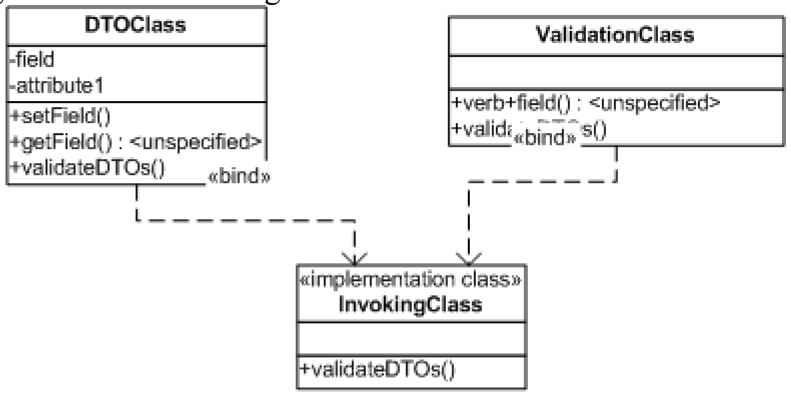
Validation of an object's state is not always required or desired. In some uses of the object, we know that the object's state is valid, as the source of the data is trusted. Flexibility of the validation logic implementation is required, as given different contexts for the use of the object; logic might be different.

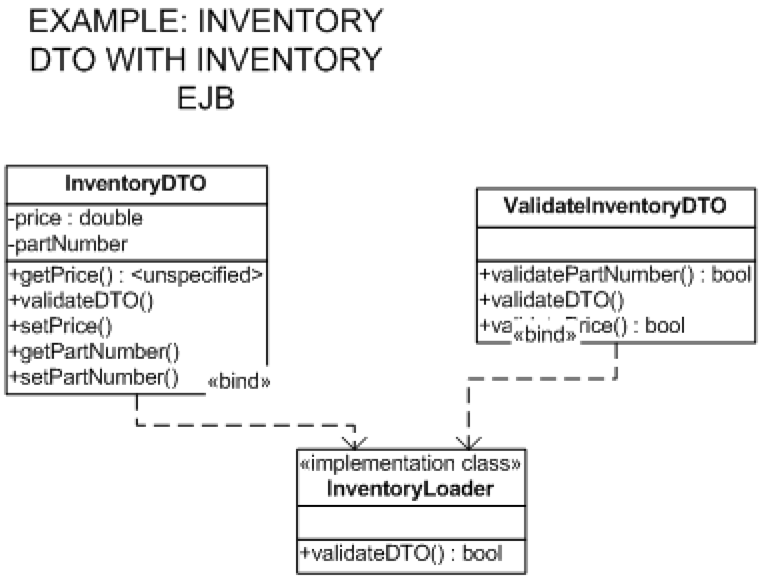
Solution

Encapsulate the validation logic into a separate class or object, away from the DTO and classes that implement it. Structure the validation logic class methods to allow for automated calls by classes using the DTO. The solution involves using the Java reflection API to recursively call the validation methods against the "getter" methods of the object's fields.

Structuring the validation logic class is accomplished by initially thinking of a simple verb that describes the actions performed against a data object. Obviously this verb is the word "validate." It is used as the prefix for each of the methods in the class that we will be calling. The suffix will be the name of the field from the DTO object. Maintaining this structure is the key to efficiently and automatically invoking, via the java reflection API, the validation logic on the DTO fields.

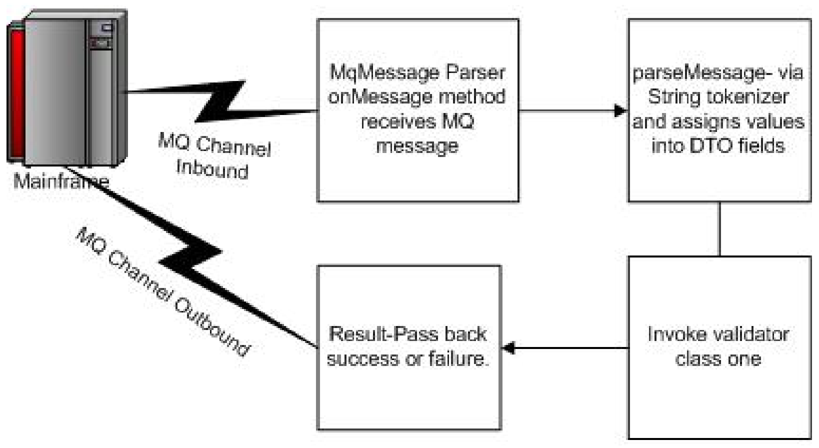
In a class where the validation logic needs to be performed on the DTO object, we add a method that uses the Java reflection API to call all of the "getter" methods of the DTO fields and the corresponding methods of the validation logic class. We then match the DTO method to the validation logic method via the naming conventions we implemented above, using the verb prefix as a key. Using this methodology, we can automate the call to business logic method onto the "getter" method.



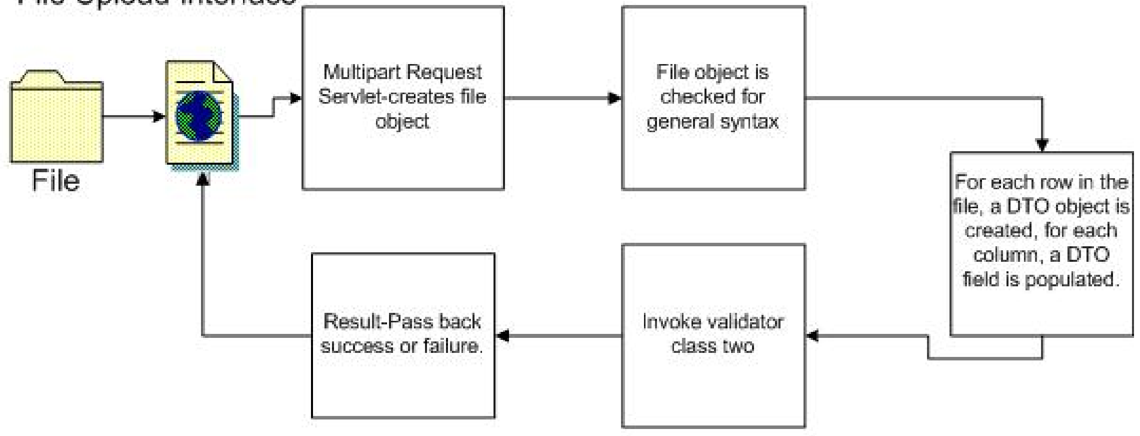


The following are examples of how we can instantiate the Validator Class using a variety of interfaces:

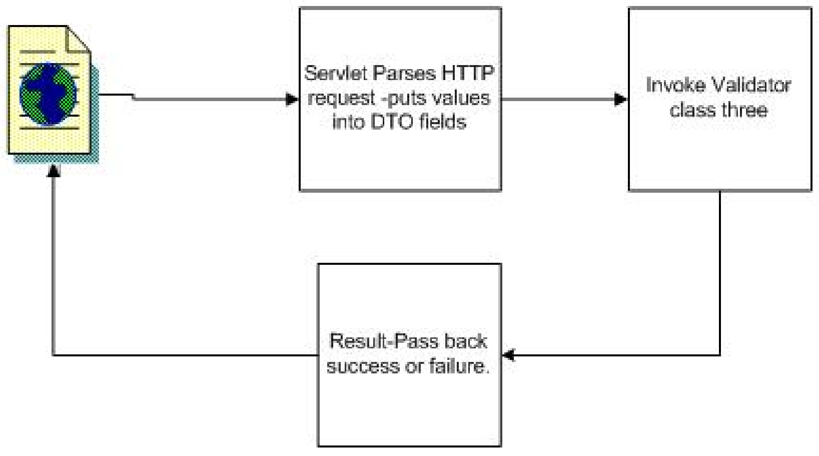
MQ Message Interface


Mainframe
MQ Channel Inbound
MQ Channel Outbound
MqMessage Parser onMessage method receives MQ message
parseMessage- via String tokenizer and assigns values into DTO fields
Invoke validator class one
Result-Pass back success or failure.

File Upload Interface


File
Multipart Request Servlet-creates file object
File object is checked for general syntax
For each row in the file, a DTO object is created, for each column, a DTO field is populated.
Invoke validator class two
Result-Pass back success or failure.

Web Interface


Servlet Parses HTTP request -puts values into DTO fields
Invoke Validator class three
Result-Pass back success or failure.

With all of the above scenarios an EJB is created if there is a success on the validation of the DTO. This way, we don't need to be concerned with rollback of transactions, or invoking an EJB "remove" method on the EJBs to clean up.

Applicability

Use the Validation Strategy when:

- You want to use an object in a variety of implementations, with differing states of the object. Validation behavior can be unique in each state of the object.

You want to automate the validation of an object's state.

Consequences

The following benefits were derived:

- **Flexibility**
  We can easily customize the implementation of an object. We can simply add a class that extends the existing validation logic class, and override the methods that we want to change. This ensures that we do not have duplicate code. In addition, if we do not need to validate an object's state, this design does not force us to do so. Furthermore, if we want to only validate certain fields, but not others (for example a "comments field"), we can do so easily.
- **Automation**
  Method calls were automated via the reflection API.

The following weaknesses occur with this pattern:

- **Limitations on Reflection API**
  The java reflection API has certain limitations; such as difficulty finding methods that are overloaded. Furthermore, this pattern depends on this API for the automated calls. Other languages will need to find other ways to call the methods against each other.
- **Method is tightly coupled to implementation class**
  The method where the validation is called could be removed from the implementation class and placed in a "proxy" class of some type.
- **Performance**
  Method calls using reflection are slower than calling the methods directly.

- **Cross Checking Validation**
  Cross checking validation will be difficult to perform.

Implementation

Implementation of this pattern is relatively straight forward, an example of which is in the Appendix. There were several issues encountered during implementation.

- Reflection based method calls to execute; we needed to be careful with the spelling and capitalization of each of the methods.

– Finding the methods is sometimes tricky. Reflection API will try and return back the first instance of a method found dynamically. If methods are overloaded, it is difficult to find the exact method with the given arguments.

Known Uses

In the August, 2002 edition of Java Pro, Mark Nadelson and Marina Evenstein, developed an architecture for automating the testing of code. Mark's framework is used to validate an object's state during testing. Rather than using a validation class as we do in this pattern, he uses serialized XML objects to perform the validation against a previously created XML "template", and recursively calls down the XML tree to validate the fields of the newly created object. When asked how he would describe the framework, he said, "I would call it the "Matching Template Object Validation Pattern."[2]

Another known use is that the pattern is similar to the old NextStep NSValidation interface, which lives on in Apple's WebObjects. See http://developer.apple.com/techpubs/webobjects/Reference/Javadoc/com/webobjects/foundation/NSValidation.html. In short, objects that implement the interface must implement a generic validateValueForKey (value, propertyName) method, which returns the validated value (which might have been coerced) or throws a ValidationException. It does this by calling the appropriate, optional, validate[Key] method if it exists. So you write the validateKey methods for each property you want to validate; the rest are considered valid by default.[4]

Related Patterns

Strategy Pattern:Validation Strategy uses the same type of structure as the Strategy Pattern. The Strategy Pattern allows a class to have different sets of behavior[3].

References

[1]Floyd Marinescu, EJB Design Patterns, The MiddleWare Company, 2002.
[2]Mark Nadelson and Marina Evenstein, "Undaunted Testing", Java Pro, vol. 6, No8, August 2002, 18-31.
[3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns*, Addison-Wesley, 1995.
[4]Ray Schnitzler, Validation of Data Transfer Objects Pattern-by Matthew A. Brown, response by Ray Schnitzler,
http://www.theserverside.com/patterns/thread.jsp?thread_id=13571

Appendix

At our company we use the Data Transfer Object (DTO) pattern for moving large amounts of data between the client and the EJB. We use IBM's Websphere Command Framework for taking requests from Java server pages (jsps) and other sources. We were writing long methods for validating the DTO's state, which resulted in less than maintainable code. We have a heterogeneous interface into our applications, including the web container (jsps and servlets); MQ Series messages, XML messaging, and file upload programs. Consequently, JavaScript does not provide the tools for validating data outside of web pages. The XML can be validated via DTD's but only to a limited extent. We wanted to standardize the way our data is validated and eliminate duplication of code. We used this pattern to perform the procedure of validating all of the fields.

DTO Class

```
package com.patterns.datatransfer;

import java.io.Serializable;
import java.sql.Timestamp;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Comparator;

public class InventoryItemDataBean
      implements Serializable, Comparator
{     private int catentryId;
      private int memberId;
      private String catentTypeId;
      private String partNumber;
      private String mfPartNumber;
      private String mfName;

      public InventoryItemDataBean()
      {}
      public int getCatentryId()
      {return catentryId;
      }
      public String getCatentTypeId()
      {return catentTypeId;
      }
      public int getMemberId()
      {return memberId;
      }
      public String getMfName()
      {return mfName;
      }
      public String getMfPartNumber()
      {return mfPartNumber;
      }
      public void setCatentryId(int newCatentryId)
      {catentryId = newCatentryId;
      }
      public void setCatentTypeId(String newCatentTypeId)
      {
      catentTypeId = newCatentTypeId;
      }
      public void setMemberId(int newMemberId)
      {memberId = newMemberId;
      }

      public void setMfName(String newMfName)
      {mfName = newMfName;
      }

      public void setMfPartNumber(String newMfPartNumber)
      {mfPartNumber = newMfPartNumber;
      }

      public void setPartNumber(String newPartNumber)
      {partNumber = newPartNumber;
      }
public InventoryItemDataBean(String argSortFieldName)
      {sortFieldName = argSortFieldName;
      }
      public int compare(Object o1, Object o2)
      { return 0;
}
      private String sortFieldName;}
```

Validation Logic Class

```
package com.patterns.validator;

import java.io.Serializable;
import java.sql.Timestamp;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Comparator;
import java.util.*;
import java.io.*;

public class InventoryItemDataValidator
{         private int memberId;
      private String catentTypeId;
      private String partNumber;
      private String mfPartNumber;
      private String mfName;



      public boolean validateCatentryId(int argCatent)
      {     return String.valueOf(argCatent).length() <5;

      }
      public boolean validateCatentTypeId(String argCatentType)
      {     return  argCatentType.equals("ItemBean");

      }

      public boolean validateMemberId(int argMemberId)
      {return true;
      }


      public boolean validateMfPartNumber(String argMfPartNumber)
      {

            return (!(argMfPartNumber == null));
      }

      public boolean validatePartNumber(String argpartnumber)
      {return true;
      }
      public InventoryItemDataValidator()
      {}
}
```

Method to initiate validation logic

1. Pass the validator and validatee classes to the method, along with the Actual DTO object.
2. Create an array of all methods for InventoryItemDataBean Data Transfer Object.
3. Filters out the "getter" methods.
4. Create an array of all of the methods of the InventoryItemDataBeanValidator that start with "validate".
5. Manipulate the name of the method validation method to see if the field name suffix matches the field name suffix of the InventoryItemDataBean "getter" method.
6. Where method prefixes, or fieldnames match, the field is validated via the reflection API.
7. For instances where validation fails, it logs a message that is returned to the requesting interface.

```
package com.patterns.facade;

import java.lang.reflect.*;
import com.patterns.validator.*;
import com.patterns.datatransfer.*;
import com.patterns.constants.ItemValidatorConstants;
/**
 * Insert the type's description here.
 * Creation date: (09/10/2002 11:49:53 PM)
 * @author: Matthew A.Brown
 */
public class InventoryItemValidatorService {
      private StringBuffer errors;
```

```
/**
 * InventoryValidator constructor comment.
 */
private InventoryItemValidatorService() {
      super();
}

  public static InventoryItemValidatorService
createInventoryItemValidatorService() {
    return new InventoryItemValidatorService();
  }
/**
 * Insert the method's description here.
 * Creation date: (09/10/2002 11:58:02 PM)
 * @param param java.lang.String
 **/
private void errorIt(String param) {
      errors.append(param);
      }
/**
 * Insert the method's description here.
 * Creation date: (09/11/2002 12:01:44 AM)
 * @return java.lang.String
 */
public String getErrors() {
      if(errors != null)
      return errors.toString();
      else
      return      "";

}
private boolean validate(com.patterns.datatransfer.InventoryItemDataBean
argdb, Class arg_Validator) {
boolean is_ok = true;
      try {
            InventoryItemDataValidator div =
(InventoryItemDataValidator)Class.forName("com.patterns.validator.Invent
oryItemDataValidator").newInstance();
            java.lang.Class InventoryItem =
Class.forName("com.patterns.datatransfer.InventoryItem");
            java.lang.Class inventoryvalidate = arg_Validator;
            java.lang.reflect.Method InventoryItem_methods[] =
InventoryItem.getMethods();
            java.lang.reflect.Method inventoryvalidate_methods[] =
inventoryvalidate.getMethods();

            for (int k = 0; k < InventoryItem_methods.length; k++) {
                  Method temp = InventoryItem_methods[k];
                  String method_name = temp.getName();
                  if (method_name.startsWith("get")) {
                        for (int l = 0; l <
inventoryvalidate_methods.length; l++) {
```

```
                              Method validator =
inventoryvalidate_methods[l];
                              String validator_name =
validator.getName();
                              if (validator_name.startsWith("validate"))
{
                                    String validate_reformat =
                                          "get" +
validator_name.substring(8, validator_name.length());
                                    if
(method_name.equalsIgnoreCase(validate_reformat)) {
                                          Object[] empty_stuff = new
Object[0];
                                          Object[] stuff = new Object[]
{temp.invoke(argdb, empty_stuff)};
                                          Boolean b = (Boolean)
validator.invoke(div, stuff);
                                          System.out.println(b.booleanVa
lue());
                                          if (!(b.booleanValue())) {
                                                is_ok = false;
                                                String results =
                                                      ("INVENTORY ITEM
WITH SPEC NUMBER:"
                                                            +
argdb.getMfPartNumber()
                                                            + "
VALIDIDATION TEST RESULTS: method calling:"
                                                            +
method_name
                                                            + "
validator calling:"
                                                            +
validator_name
                                                            + "
ResultOfTest: "
                                                            + b
                                                            + "<BR>\n");
                                                System.out.println(resul
ts);
                                                errorIt(results);
                                                return false;
                                          } //IF
                                    } //IF METHOD
                              } //IF VALIDATOR
                        }
                  }
            }
      } catch (Exception e) {
            System.out.println(e);
 }
      return is_ok;
```

```java
}
/**
 * Insert the method's description here.
 * Creation date: (09/11/2002 12:17:11 AM)
 * @return java.lang.String
 */
public boolean validateItem(InventoryItemDataBean arg_inventory_item,
int arg_interface) {
      boolean validated = false;
      try{
        if(arg_interface ==
(com.patterns.constants.ItemValidatorConstants.FILE_INTERFACE))
         validated =
(validate(arg_inventory_item,Class.forName("com.patterns.validator.Inven
toryItemDataValidatorForFiles")));
           else if(arg_interface ==
(com.patterns.constants.ItemValidatorConstants.JMS_INTERFACE))
             validated =
(validate(arg_inventory_item,Class.forName("com.patterns.validator.Inven
toryItemDataValidatorForJMS")));
              else
              validated =
(validate(arg_inventory_item,Class.forName("com.patterns.validator.Inven
toryItemDataValidatorForWeb")));
      }catch(Exception e){
            e.printStackTrace();
      }
      return validated;
      }
}
```

Constants

```java
package com.patterns.constants;

/**
 * Insert the type's description here.
 * Creation date: (09/11/2002 12:20:25 AM)
 * @author: Matthew A. Brown
 */
public class ItemValidatorConstants {
      public final static int JMS_INTERFACE = 0;
      public final static int WEB_INTERFACE = 1;
      public final static int FILE_INTERFACE = 2;
/**
 * ItemValidatorConstants constructor comment.
 */
public ItemValidatorConstants() {
      super();
}
}
```