Data Validation 1.1 Component Specification

The Data Validation component defines an ObjectValidator interface with a collection of default implementations, which provide convenient and complex functionality for data validation. These implementations can be retrieved from factory methods that cover the primitive data types and their wrapper class. Example Validators include a range-validator, whitespace-validator, and substring-validator. A primitive Validator can validate its own data type, other data types that can be converted to its data type or strings that can be parsed into its data type.
Primitive Validators are abstract and intended to be extended.
The `PrimitiveValidator` is used to wrap an ObjectValidator into a Validator that can validate any java data type. This is achieved by providing overridden methods for each primitive data type and wrapping the primitive value in its object class before calling the underlying Validator.
The `TypeValidator` is used to validate a given object of a specified type. It can also wrap an underlying Validator to ease construction overhead.
The component also contains composite validators such as AndValidator, OrValidator and `NotValidator`. This allows construction of complex validation logic with little effort.

Please note that changes to version 1.1 will be shown in **red** (new entries) and **blue** (modification of existing entries).

1. Design

1.1.1 Base implementations decisions

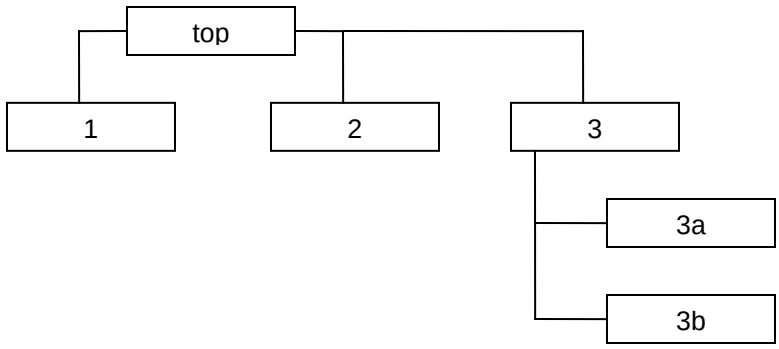
The following decisions have been made as far as requirements are concerned.

1.1.1.1 In version 1.1, support for returning validation messages as a collection of Strings is added. This way, the component is more flexible for GUI interfaces to format and display validation error messages.

This is basically a way of collecting all the messages from all relevant validators that make up our composite validator. Thus if the user has a composite validator the following API would be utilized

```
String[] getMessages(Object obj);
```

This means that if the validator is composite we will get all the messages if the validation failed. Here is how we can imagine a typical composite validator:



In the above diagram we can imagine that the top validator is an AndValidator and the other validators are all either primitive (leaves) or composite (nodes). The point of the diagram is to show that, potentially, when calling the new required API we would need to remember each message from the failed leaves. Let consider the following validation path <top, 3, 3a, 3b> and lets say that the 3a and 3b validators both fail (we can assume that 3 is an `OrValidator`) this means that we would have the following failure trace: <fail, fail, fail, fail> and it could mean that

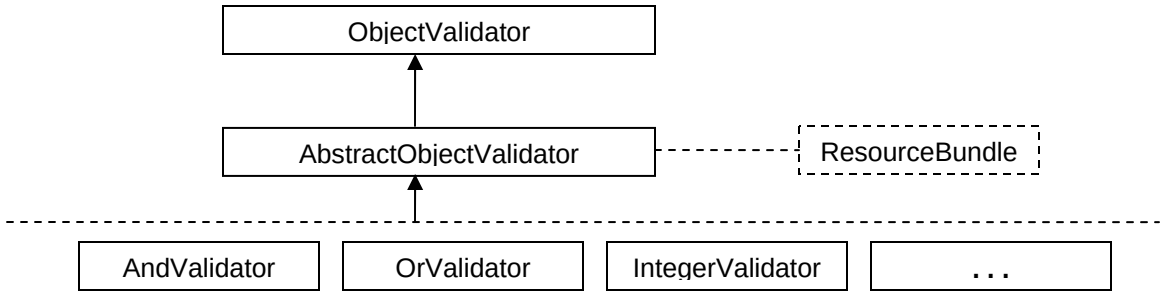
we should get a message from each leaf which in this case means that we should get back an array with 3a-message and 3b-message. In other words as per the requirement we only return the messages from the failed leaves.

Also note that the evaluation of any hierarchy is by default short-circuited, which means that the moment the result is known the sub-hierarchy will not be evaluated and thus additional messages will not be added. Consider this path of validation: <top, 1> if the 1-validator fails there is no point to evaluate 2 or 3 (assuming that top is an `AndValidator`) even though 3 might still have all of its sub-leaves fail the input.

All of this will be achieved by the assumption that any composite validator will only return the messages created by its nodes and we will simply keep on adding these messages to an internal list which eventually will hold all the needed messages.

1.1.1.2 Support for internationalization is required in this new version without loosing backward compatibility.

The main idea behind the 1.1 component architecture is to create a layer of abstraction between all the current validator implementations and the main `ObjectValidator` interface that they all implement. This in essence will create a layer of support, which will include internationalization (i.e. resource bundling) for all currently descended validators. Here is what it would look like:



The basic idea here is that the AbstractObjectValidator will supply all the internationalization functionality directly to the descendants who previously just extended/implemented ObjectValidator.

The way that this will be implemented is that we will create the following constructors for `AbstractObjectValidator`:

```
AbstractObjectValidator(BundleInfo bundleInfo)
```

where the following information is contained in the BundleInfo class

- bundleName is the name of the resource Bundle to use
- locale is the locale to use
- messageKey is the key to use to obtain the message from the bundle
- defaultMessage is the message to use if there was a failure to load up the message from the resource bundle.

Note that the bundle information will always be copied so the original can be conveniently reused.

1.1.1.3 Serialization of Validators

As required all validators will be serializable. This is achieved by simply having the ObjectValidator interface be extended from Serializable interface. Note that for efficiency reasons the ResourceBundle will be loaded up from the provided name upon construction but since ResourceBundle is itself NOT serializable we will need to ensure that we properly serialize/deserialize the bundle which means that the internal resourceBundle variable will have to be transient as follows:

```
private transient ResourceBundle resourceBundle
```

We will also need to be able to properly serialize the variable and we will need these for that:

```
private  void writeObject(java.io.ObjectOutputStream stream) throws
    java.io.IOException {
        stream.defaultWriteObject(  );
}

private  void readObject(java.io.ObjectInputStream stream) throws
    java.io.IOException {
        stream.defaultReadObject(  );
        // load up the bundle
        if(currentLocale == true){
            bundleLocale = // current locale
        }
        resourceBundle =
            ResourceBundle.getBundle(bundleName, bundleLocale);
}
```

1.1.2 General changes from version 1.0 to 1.1

There are a number of things that will be changed:

1. The version 1.1 design will follow all TopCoder coding conventions as long as their introduction will not change existing client code. Thus the following aspects will be changed
    a. All protected variables will be changed to private (and final where applicable) and if necessary get/set or other accessor/mutator methods will be introduced.
2. Documentation and presentation will be brought up to the current TopCoder standards.
    a. Class and method documentation will be brought up to the current TopCoder standards
    b. UML will be brought up to the current TopCoder standards.
    c. The CS document will follow the latest TopCoder document template.

1.1.3 Functional Enhancements

Here we are listing the enhancements that have been added to this component on top of what the requirements have asked for.

1.1.3.1 Extra validators

The following validators have been added to the design as being deemed to be helpful:

1. RegexpStringValidator, which is a validator that allows for regular expression validation. This was painfully missing from the StringValidator repertoire.

1.1.3.2 Non short circuited composite validator evaluation

Currently the new version of valid method, which returns multiple validation messages is short-circuited which means that for an AndValidator only one branch needs to be tested to see if the result is false and if so other branches will not be tested, and for an OrValidator we would have

the same with a branch that returns true. This can be sometimes a drag since it would be nice to know all the errors regardless of the composition of the validator. This enhancement will provide exactly such API with an additional convenience, which will limit the number of returned messages. Here is what the API will look like:

```
String[] getAllMessages(Object obj, int messageLimit);
String[] getAllMessages(Object obj);
```

If the `messageLimit` is used it will limit the number of returned messages and if the other overloaded version is used it will return all the messages and will evaluate every branch.

1.1.4 API enhancements

The following aspects of the 1.0 version API has been enhanced in version 1.1

1. `AbstractAssociativeObjectValidator` class has now the ability to remove associated validators as well as list/iterate over existing iterators.
2. All of the numeric based validators such as `ByteValidator`, `LongValidator`, etc… were missing the notion of equality validation. This was added to their API as well as to the `CompareDirection` enumeration (we added the EQUAL direction)

1.2 Design Patterns

Composite Pattern: The component uses the composite pattern to allow several validators to work together in “and/or“ mode. This facilitates a more natural representation of validation logic.
Factory Method: The component uses a factory method to concentrate similar functionality for primitive data types into one class. This makes both simplifies the package structure and component usage.

1.3 Industry Standards

Java 1.4.x
I18N

1.4 Required Algorithms

Please note that complete source code has been provided in the \docs\prototype_src directory and the developer should consult that for their algorithm needs.

1) The AbstractAssociativeObjectValidator class implements the add validator logic. This prevents the AndValidator and OrValidator classes from having to implement it multiple times.

2) For the getMessage(Object) or valid(Object) The AndValidator class should return true when it is empty. Likewise, the class OrValidator should return false (or the non-null message) when it is empty. Here is the general loop logic:

```
AndValidator.valid(Object)
    for (Iterator itr = getValidators(); itr.hasNext();){
        if (!((ObjectValidator) itr.next()).valid(obj)){
                return false;
        }
   }
    return true;


OrValidator.valid(Object)
    for (Iterator itr = getValidators(); itr.hasNext();) {
        if (((ObjectValidator) itr.next()).valid(obj)){
            return true;
        }
    }
    return false;
```

3) There is an abstract class for each primitive data type, including java.lang.String. Each of these classes has two “valid” methods, one for objects and one for the primitive data type. The primitive Validators should implement the Object version; converting the primitive to object form and calling the object validate method. The other method is left abstract. In this way the user only has to extend the primitive validator and implement the second method and immediately has the conversion logic.

4) Conversion should be carried out as follows:

```
IntegerValidator.valid(Object)
    if (obj instanceof Number) {
        return valid(((Number) obj).intValue());
    } else (obj instanceof String) {
        try {
            return valid(Integer.parseInt((String) obj));
        } catch (Exception exception) {
    }
    return false;
```

5) The two floating point data types provide a facility to account for epsilon. The default implementation is “YES”. Epsilon can also be set by a call to the setEpsilon method. This will overwrite the default value (which is 1e-8 for float and 1e-12 for double). The user can also set epsilon to a non-positive value which means epsilon should not be considered.

6) The Validators returned from the factory methods take conversion into consideration. Type-strong validation can be achieved by using the TypeValidator class. This class acts as a wrapper for the validator that first validates on the type of the Object. It can also be used without an underlying validator, meaning it is only interested in the type of the object.

7) The validators returned by the factory methods can be implemented with inner class, package class or anonymous class. It is to make use of some common validation mechanisms. For example the IntegerValidator.isPositive() method could be just one line: return IntegerValidator.greaterThan(0);

Some sample private/inner class definitions:

```
class IntegerValidatorWrapper extends IntegerValidator {
    private ObjectValidator validator = null;
        public MyIntegerValidatorWrapper(ObjectValidator validator) {
            this.validator = validator;
        }
        public boolean valid(Object obj) {
            return validator.valid(obj);
        }
        public boolean valid(int value) {
            return validator.valid(new Integer(value));
        }
}


class CompareIntegerValidator extends IntegerValidator {
    private int value;
    boolean isLess = true;

    public CompareIntegerValidator(int value, boolean isLess) {
        this.value = value;
        this.isLess = isLess;
    }

    public boolean valid(int value) {
        return isLess ? value < this.value : value > this.value;
    }
}
```

```
class GreaterIntegerValidator extends IntegerValidator {
    private int value;
        public LessIntegerValidator(int value) {
           this.value = value;
        }

        public boolean valid(int value) {
           return value > this.value;
        }
}
```

Some sample one liner p[roposed method implementations:

```
IntegerValidator.lessThan(int value):
        return new CompareIntegerValidator(value, true);

IntegerValidator.greaterThan(int value)
        return new CompareIntegerValidator(value, false);

IntegerValidator.lessThanOrEqualTo(int value):
        return new IntegerValidatorWrapper(
                        new NotValidator( new CompareValidator(value, false)));

IntegerValidator.greaterThanOrEqualTo(int value):
        return new IntegerValidatorWrapper(
                new NotValidator( new CompareValidator(value, true)));

IntegerValidator.inRange(int lower, int upper):
        return new IntegerValidatorWrapper(
                new AndValidator( greaterThanOrEqualTo(lower)
                        , lessThanOrEqualTo(upper))));

IntegerValidator.inExclusiveRange(int lower, int upper):
        return new IntegerValidatorWrapper(
                new AndValidator( greaterThan(lower)
                        , lessThan(upper))));

IntegerValidator.isPositive():
        return new CompareIntegerValidator(0, false);

IntegerValidator.isNegative():
        return new CompareIntegerValidator(0, true);
```

8) All CharaterValidator.isXXX() methods reference the Character.isXXX() method. They are semantically the same.

9) The component also provides a PrimitiveValidator to validate “anything”. This is constructed with an ObjectValidator. Methods for each primitive data type would wrap the value in its wrapper class and call the underlying validator.

10) The ObjectValidator interface also declares a getMessage(Object) method. This method parallels the valid(Object) method. This method provides an explanation as to why the object in question fails validation. However, this method can replace the valid(Object) method since it should return null when the object actually validates. But the boolean method is still in place and it is more natural and convenient when you an explanation is not required.

11) When it comes to implementing the `getMessages(Object)` method this is quite simple since all we need to do here is call the `getMessages(Object)` and retain the messages coming from any failed validator. This is quite simple and the loop logic is exactly the same as in getMessage(Object) This really only affects the OrValidator logic (the AndValidator not being changed and the rest being trivial) here is what needs to be done in the loop:

```
OrValidator.getMessages(Object)
        List allMessages = new ArrayList();

        // Iterate all validators
        for (Iterator itr = getValidators(); itr.hasNext();) {
            String[] messages = ((ObjectValidator) itr.next()).getMessages(object);

            if (messages == null) {
        // Validation successful
        return null;
            } else {
        for (int i = 0; i < messages.length; i++) {
            allMessages.add(messages[i]);
        }
            }
        }

        return (String[]) (allMessages.toArray(new String[0]));
```

11) The ObjectValidator interface defines the non-short-circuited version of the getMessages(Object) method which is the getAllMessages(…) which comes in two flavors, one which returns all the messages and one which returns an upper bound of messages. Here is how the upper bound of the OrValidator version would work (note that AndValidator version is basically the same logic) The most important aspect is to be able to track the number of currently obtained messages (which starts a 0). This is done by sending to each validator a modifiable `currentCount` parameter, which is used to track the current number of encountered messages:

```
OrValidator.getAllMessages(Object, int limit, int currentCount) {
        List allMessages = new ArrayList();

        for (Iterator itr = getValidators(); itr.hasNext() && (allMessages.size() <
        messageLimit);) {
              String[] messages = ((ObjectValidator) itr.next()).getAllMessages(object,
        messageLimit - allMessages.size());

              if (messages == null) {
                  return null;
              } else {
                  // Only add if current number of messages is less than messageLimit
                  for (int i = 0; (i < messages.length) && (allMessages.size() <
        messageLimit); i++) {
                        allMessages.add(messages[i]);
                  }
              }
        }

        return (String[]) (allMessages.toArray(new String[0]));
}
```

1.5 Component Class Overview

1.5.1 com.topcoder.util.datavalidator

AbstractAssociativeObjectValidator

This is am abstract class that is used to abstract functionality of validator composition. It is basically a composition of 0 or more validators. This class gives its descendants the ability to manipulate the aspects of validator composition. The validators are guaranteed to be placed in a normal list order which means that the first validator added to this composition will be the first that is returned through the iterator, the second will be the second one returned, etc...

AbstractObjectValidator

This is an abstract ObjectValidator implementation, which gives the ability for resource bundled message fetching for the validation messages. The user will be able to set up which bundle should be used and what key should be used to fetch the message. We also have the ability to specify the locale.
But if one is not provided then the current locale will be used. Note that the bundle functionality is fail-safe in the sense the user must provide a default message in case the bundle functionality fails. This way the validator is always guaranteed to be able to produce a message. The user is not required though to utilize the resource bundle functionality and can directly supply just the message that will be then used by the implementing validator.

AlphanumCharacterValidator
This is a specific type of CharacterValidator, which will validate an input character as a digit or a letter (but not both) It can act in one of those modes:

1. Check of the input character is a digit
2. Check if the input character is letter

This is determined by the initialization done in the constructor, which allows the user to choose either a digit validation (isDigit - true) or letter validation (idDigit - false)
This class is thread-safe since it is immutable. Note that this is an inner class of the CharacterValidator and is not part of the public API.

AndValidator
This is a specific type of a composite validator which validates based on a boolean AND outcome of all the associated validators. This is basically a composite AND validator. By default it works in a short-circuited mode which means that the validator will return an 'invalid' result the moment the first of the validators returns a false (or null for getMessage())

BooleanValidator
This is a simple boolean validator abstraction which basically checks that a given boolean value conforms with the state of this validator. This validator is abstract and will need to be extended to actually provide the validation routine. One such example would be providing a boolean value validation which fails validation of the provided object evaluates to false. User will need to implement the `valid(boolean value)` method to decide what the validator will do.

ByteValidator
This is a simple byte validator abstraction, which basically checks that a given value conforms with a specific byte definition. This validator is abstract and will need to be extended to actually provide the validation routine for a specific byte value validation. Note that this validator acts as a factory that allows for creation of other specialized ByteValidator instances that provide a number of convenience methods which validate such aspects as range (from, to and could also be exclusive of extremes) or provide general comparison of values such as greater-than, less-than, etc...
User will need to implement the `valid(byte value)` method to decide what the validator will do with the input byte.

ByteValidatorWrapper
This is a simple byte validator abstraction, which basically checks that a given value conforms to a specific byte definition. This validator is abstract and will need to be extended to actually provide the validation routine for a specific byte value validation. Note that this validator acts as a factory that allows for creation of other specialized ByteValidator instances that provide a number of convenience methods which validate such aspects as range (from, to and could also be exclusive of extremes) or provide general comparison of values such as greater-than, less-than, etc...

User will need to implement the `valid(byte value)` method to decide what the validator will do with the input byte.

CaseCharacterValidator
This is a specific type of CharacterValidator, which ensures either the upper case, or lower case aspect of input (i.e. of the character). Thus this validator works in one of two modes:

1. Test for upper case
2. Test for lower case

User will choose which mode it operates in through the constructor flag. This is thread-safe since it is immutable. This is an inner class of `CharacterValidator`

CharacterValidator
This is a simple character validator abstraction, which basically checks that a given value conforms to a specific character definition. This validator is abstract and will need to be extended to actually provide the validation routine for a specific character value validation. Note that this validator acts as a factory that allows for creation of other specialized CharacterValidator instances that provide a number of convenience methods, which validate such aspects as if the character is a digit or a letter or both, if it is a white space or if it is upper-case or lower-case.
User will need to implement the valid`(char value)` method to decide what the validator will do with the input character.

CharacterValidatorWrapper
This is something of an adapter class, which gives us a simple implementation of the abstract CharacterValidator class. Note that this will wrap around any type of a validator but since this is an internal class only CharacterValidator class is expected. It is thread-safe as it is immutable. This is an inner class to `CharacterValidator`.

CompareByteValidator
This is an extension of the ByteValidator, which specifically deals with validating comparisons based criteria such equal, greater-than-or-equal, less-than, less-than-or-equal, greater-than. Thus we will have a validation based on a specific initialization value (that is set with the validator) and then have this validator compare the input value with the set value using one of the specific comparison directions. This is an inner class to ByteValidator. This class is thread-safe as it is immutable.

CompareDirection
This is a simple enumeration of comparison directions that we would see when comparing entities such as numbers. All the possible directions are enumerated here such as equal, greater-than-or-equal, less-than-or-equal, greater-than, and less-than. As an enumeration it is thread-safe since it is immutable.

CompareDoubleValidator
This is an extension of the `DoubleValidator`, which specifically deals with validating comparisons based criteria such equal, greater-than-or-equal, less-than, less-than-or-equal, greater-than. Thus we will have a validation based on a specific initialization value (that is set with the validator) and then have this validator compare the input value with the set value using one of the specific comparison directions. This is an inner class to DoubleValidator. This class is thread-safe as it is immutable.

CompareFloatValidator

This is an extension of the FloatValidator, which specifically deals with validating comparisons based criteria such equal, greater-than-or-equal, less-than, less-than-or-equal, greater-than. Thus we will have a validation based on a specific initialization value (that is set with the validator) and then have this validator compare the input value with the set value using one of the specific comparison directions. This is an inner class to `FloatValidator`. This class is thread-safe as it is immutable.

CompareIntegerValidator

This is an extension of the `IntegerValidator`, which specifically deals with validating comparisons based criteria such equal, greater-than-or-equal, less-than, less-than-or-equal, greater-than. Thus we will have a validation based on a specific initialization value (that is set with the validator) and then have this validator compare the input value with the set value using one of the specific comparison directions. This is an inner class to `IntegerValidator`. This class is thread-safe as it is immutable.

CompareLongValidator

This is an extension of the `LongValidator`, which specifically deals with validating comparisons based criteria such equal, greater-than-or-equal, less-than, less-than-or-equal, greater-than. Thus we will have a validation based on a specific initialization value (that is set with the validator) and then have this validator compare the input value with the set value using one of the specific comparison directions. This is an inner class to `LongValidator`. This class is thread-safe as it is immutable.

CompareShortValidator

This is an extension of the `ShortValidator`, which specifically deals with validating comparisons based criteria such equal, greater-than-or-equal, less-than, less-than-or-equal, greater-than. Thus we will have a validation based on a specific initialization value (that is set with the validator) and then have this validator compare the input value with the set value using one of the specific comparison directions. This is an inner class to ShortValidator. This class is thread-safe as it is immutable.

ContainsStringValidator

This is an extension of the `StringValidator`, which specifically deals with validating strings based on substring containment. This means that we will be validating user input as being valid it is contains the string that this validator has been initialized with. As an example, if we create an instance of this validator set with a string of "hello" then any user input that would contain this string (case-insensitive) would be considered valid. This is an inner class to StringValidator. This class is thread-safe as it is immutable.

DoubleValidator

This is a simple double validator abstraction, which basically checks that a given value conforms to a specific double definition. There is also an option to use this validator with an error of comparison delta which is termed here as epsilon which means that comparisons could be done to within this particular epsilon. This validator is abstract and will need to be extended to actually provide the validation routine for a specific double value validation. Note that this validator acts as a factory that allows for creation of other specialized DoubleValidator instances that provide a number of convenience methods which validate such aspects as range (from, to and could also be exclusive of extremes) or provide general comparison of values such as greater-than, less-than, etc... We can also create validators that validate for negative or positive values or test if the double can be considered to be an integer (i.e. no loss of precision when converting or with specified and acceptable loss of precision) User will need to implement the `valid(double value)` method to decide what the validator will do with the input byte. This is thread-safe as the implementation is immutable.

DoubleValidatorWrapper

This is something of an adapter class, which gives us a simple implementation of the abstract DoubleValidator class. Note that this will wrap around any type of a validator but since this is an internal class only `DoubleValidator` class is expected. This is an inner class to DoubleValidator. It is thread-safe as it is immutable.

EndsWithStringValidator
This is an extension of the `StringValidator`, which specifically deals with validating strings based on testing that strings end with a specific string. As an example, if we create an instance of this validator set with a string of "world" then any user input that would end with this string (case-insensitive) would be considered valid. This is an inner class to `StringValidator`. This class is thread-safe as it is immutable.

FloatValidator
This is a simple float validator abstraction, which basically checks that a given value conforms to a specific float definition. There is also an option to use this validator with an error of comparison delta which is termed here as epsilon which means that comparisons could be done to within this particular epsilon. This validator is abstract and will need to be extended to actually provide the validation routine for a specific double value validation. Note that this validator acts as a factory that allows for creation of other specialized FloatValidator instances that provide a number of convenience methods which validate such aspects as range (from, to and could also be exclusive of extremes) or provide validation of general comparison of values such as greater-than, less-than, etc... We can also create validators that validate for negative or positive values or test if the float can be considered to be an integer (i.e. no loss of precision when converting or with specific acceptable loss of precision) User will need to implement the `valid(float value)` method to decide what the validator will do with the input byte. This is thread-safe as the implementation is immutable.

FloatValidatorWrapper
This is something of an adapter class, which gives us a simple implementation of the abstract `FloatValidator` class. Note that this will wrap around any type of a validator but since this is an internal class only `FloatValidator` class is expected. This is an inner class to FloatValidator. It is thread-safe as it is immutable.

IntegerValidator
This is a simple integer validator abstraction, which basically checks that a given value conforms to a specific integer definition. This validator is abstract and will need to be extended to actually provide the validation routine for a specific integer value validation. Note that this validator acts as a factory that allows for creation of other specialized `IntegerValidator` instances that provide a number of convenience methods which validate such aspects as range (from, to, and could also be exclusive of extremes) or general comparison of values such as greater-than, less-than, etc... We can also create validators, which validate if a number is positive or negative, or even if it is odd or even (i.e. parity comparison) User will need to implement the `valid(int value)` method to decide what the validator will do with the input integer. This is thread-safe as the implementation is immutable.

IntegerValidatorWrapper
This is something of an adapter class, which gives us a simple implementation of the abstract `IntegerValidator` class. Note that this will wrap around any type of a validator but since this is an internal class only IntegerValidator class is expected. This is an inner class to `IntegerValidator`. It is thread-safe as it is immutable.

IntegralDoubleValidator

This is an extention of the `DoubleValidator` which specifically deals with validating that double values are actually convertible to integers without loss of precision (or with acceptable epsilon loss of precision) In other words this validates if the input is a mathematical integer. Note that if epsilon is set it will be used in the comparison, which means that we will be accepting an approximation of the integer. This is an inner class to `DoubleValidator`. This class is thread-safe as it is immutable.

IntegralFloatValidator

This is an extension of the `FloatValidator` which specifically deals with validating that float values are actually convertible to integers without loss of precision (or with acceptable epsilon loss of precision) In other words this validates if the input is a mathematical integer. Note that if epsilon is set it will be used in the comparison, which means that we will be accepting an approximation of the integer. This is an inner class to `FloatValidator`. This class is thread-safe as it is immutable.

LengthStringValidator

This is an extension of the `StringValidator`, which specifically deals with validating strings based on their length. Note that the actual validation is performed based on an `IntegerValidator` which means that we could have this validator do all the integer validations including range validation of the length. As an example, if we create an instance of this validator set with an `IntegerValidator` which checks that a number is in range of 2..10 inclusive then any user input that would have its length in that range would be considered valid. This is an inner class to `StringValidator`. This class is thread-safe as it is immutable.

LongValidator

This is a simple long integer validator abstraction, which basically checks that a given value conforms to a specific long integer definition. This validator is abstract and will need to be extended to actually provide the validation routine for a specific long integer value validation. Note that this validator acts as a factory that allows for creation of other specialized `LongValidator` instances that provide a number of convenience methods which validate such aspects as range (from, to, and could also be exclusive of extremes) or general comparison of values such as greater-than, less-than, etc... We can also create validators that validate if a number is positive or negative, or even if it is odd or even (i.e. parity comparison) User will need to implement the `valid(long value)` method to decide what the validator will do with the input byte. This is thread-safe as the implementation is immutable.

LongValidatorWrapper

This is something of an adapter class, which gives us a simple implementation of the abstract `LongValidator` class. Note that this will wrap around any type of a validator but since this is an internal class only `LongValidator` class is expected. This is an inner class to `LongValidator`. It is thread-safe as it is immutable.

NotValidator

This is a specific type of a composite validator which validates based on a boolean NOT outcome of the associated validator. In other words it reverses the validation outcome of the associated validator. Note that the message returned by this validator in case of failure doesn't take into account the associated validator since it reverses the successful validation outcome of that validator. This class is thread-safe as it is immutable.

NullValidator

This is a specific type of a simple validator which validates based on a boolean whether the input object is null or not. This class is thread-safe as it is immutable.

ObjectValidator

This is the main contract for object validation. Here we have the ability for simple validation (true/false), which will tell us if the object input is valid, or not. We also have the ability to specify a validation that will produce a message if the validation fails telling us something about the cause. For complex validation where we could have multiple failure points we have the ability to fetch all the messages that come from the validators that failed the object. This can further be refined into a short-circuited evaluation or the non-short-circuited evaluation. In short-circuited evaluation we have a scenario where the validator will return a failed result (with associated) messages, the moment it can determine that the validation has failed. For example and `AndValidator` need only fail the first of its constituent validators to know that the whole validation fails. Thus there is no need to invoke any further validators in the tree. Thus we have short-circuited `AndValidator` for failure. The same way the `OrValidator` is short-circuited for success. In non-short-circuited evaluation we keep on traversing the validator tree regardless of the fact that the composite outcome is already determined. This is good for situations where we would like to return to the user as much as possible information so that they can correct the issues as in a web page input for example. There is also a variation on this with being able to limit the number of messages that are returned to the caller so as to put an upper limit on the validator tree traversal. Implementations of this are required to be thread-safe.

OrValidator
This is a specific type of a composite validator which validates based on a boolean OR outcome of all the associated validators. This is basically a composite OR validator. By default it works in a short-circuited mode which means that the validator will return a 'valid' result the moment the first of the validators returns a true (or null for `getMessage()`) This class is thread safe since it adds no shared/mutable data

ParityIntegerValidator
This is an extension of the IntegerValidator which specifically deals with validating numbers based on their parity (i.e. if the number is odd or even) Which parity should be validated against is set as a flag when creating/initializing this validator and is basically odd (true) or even (false) As an example, if we create an instance of this validator set with a parity of odd (true) which would then accept only integers that are odd. This is an inner class to `IntegerValidator`. This class is thread-safe as it is immutable.

ParityLongValidator
This is an extension of the LongValidator which specifically deals with validating numbers based on their parity (i.e. if the number is odd or even) Which parity should be validated against is set as a flag when creating/initializing this validator and is basically odd (true) or even (false) As an example, if we create an instance of this validator set with a parity of odd (true) which would then accept only long integers that are odd. This is an inner class to LongValidator. This class is thread-safe as it is immutable.

PrimitiveValidator
This is a specific wrapper for all validators that deal with primitive type value validation such a `Integer, Boolean, Long, Double, Float, Byte,` or `Short` this is nothing more than a convenience wrapper. This is thread-safe since it is immutable.

RangeDoubleValidator
This is an extension of the `DoubleValidator`, which specifically deals with validating a range. In other words we are creating a validator that will validate if a given input number is in range (inclusive of the extremes as an option) Note that an acceptable comparison error can be set as well (i.e. the epsilon) which would be taken into account when deciding if the in-range criteria has been met. As an example, if we create an instance of this validator set with a range of 2.5 - 7.88 with an epsilon of 1e-12 then a number like 5.0

would be in range and even a number like 7.88000000045 would be in range but a number like 7.89 would not be valid. This is an inner class to `DoubleValidator`. This class is thread-safe as it is immutable.

RangeFloatValidator
This is an extension of the `FloatValidator`, which specifically deals with validating a range. In other words we are creating a validator that will validate if a given input number is in range (inclusive of the extremes as an option) Note that an acceptable comparison error can be set as well (i.e. the epsilon) which would be taken into account when deciding if the in-range criteria has been met. As an example, if we create an instance of this validator set with a range of 2.5f - 7.88f with an epsilon of 1e-12 then a number like 5.0 would be in range and even a number like 7.88000000045 would be in range but a number like 7.89f would not be valid. This is an inner class to `FloatValidator`. This class is thread-safe as it is immutable.

RegexpStringValidator
This is an extension of the `StringValidator`, which specifically deals with validating strings based on a regular expression. This is an inner class to StringValidator. This class is thread-safe as it is immutable.

ShortValidator
This is a simple short integer validator abstraction, which basically checks that a given value conforms to a specific short integer definition. This validator is abstract and will need to be extended to actually provide the validation routine for a specific short integer value validation. Note that this validator acts as a factory that allows for creation of other specialized ShortValidator instances that provide a number of convenience methods which validate such aspects as range (from, to and could also be exclusive of extremes) or general comparison of values such as greater-than, less-than, etc... Users will need to implement the `valid(short value)` method to decide what the validator will do with the input short integer. This is thread-safe as the implementation is immutable.

ShortValidatorWrapper
This is something of an adapter class, which gives us a simple implementation of the abstract `ShortValidator` class. Note that this will wrap around any type of a validator but since this is an internal class only `ShortValidator` class is expected. This is an inner class to `ShortValidator`. It is thread-safe as it is immutable.

StartsWithStringValidator
This is an extension of the `StringValidator`, which specifically deals with validating strings based on testing that strings start with a specific string. As an example, if we create an instance of this validator set with a string of "hello" then any user input that would start with this string (case-insensitive) would be considered valid. This is an inner class to `StringValidator`. This class is thread-safe as it is immutable.

StringValidator
This is a simple String validator abstraction, which basically checks that a given value conforms to a specific string definition. This validator is abstract and will need to be extended to actually provide the validation routine for a specific string value validation. Note that this validator comes with a number of convenience factory methods, which create specific valiators, which can tests/validate such aspects as if a string starts with, or end with some string. We also have the ability to create validators that will test for string containment (i.e. a string is valid if it contains a certain substring) or validators that test for length based comparisons such as exact length or a range. We also have the ability to

create validators that test an input string against specific regular expression. User will need to implement the `valid(String value)` method to decide what the validator will do with the input String. This is thread-safe as the implementation is immutable.

TypeValidator
This is a simple `ObjectValidator`, which specifically deals with validating the input type of a value. It disregards all primitive based class types (such as `Double` for example) and considered any input that is primitive based wrapper to be invalid. User can create this validator with a specific class type and then it (or its descendants) would be considered as the valid type to be used by this validator. This class is thread-safe as it is immutable.

WhitespaceCharacterValidator
This is an extension of the StringValidator which specifically deals with validating strings based on whether they are to be consider a white space or not. This is an inner class to `StringValidator`. This class is thread-safe as it is immutable.

BundleInfo
This is a simple bean, which represents the resource bundle information for validators. This is a serializable bean and is not thread-safe. This doesn't affect the thread-safety of the component though since it will be utilized in a thread-safe manner.

1.6 Component Exception Definitions

1.6.1 Custom Exceptions

None provided.

1.6.2 System exceptions

- **IllegalArgumentException**: Exception thrown in various methods where value is invalid or null. This could be because some passed in value is NAN, or null or out of allowable range for example.

1.7 Thread Safety

As required for this component the validator implementations are all thread safe. This is achieved by having most classes being immutable and where there is shared modifiable data (such as the `AbstractAssociativeObjectValidator` validator list) we simply synchronize around read/write access to this data.
The only exception is the `BundleInfo` bean, which is mutable. But it is utilized in an immutable manner by making sure that a copy is always used and returned to the user so that there are no issues with multiple threads accessing this bean.

2. Environment Requirements

2.1 Environment

- Development language: Java 1.4
- Compile target: Java 1.4

2.2 TopCoder Software Components

None used.

2.3 Third Party Components

None used.

3. Installation and Configuration

3.1 Package Name

com.topcoder.util.datavalidator

3.2 Configuration Parameters

None.

3.3 Dependencies Configuration

None

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute 'ant test' within the directory that the distribution was extracted to.

4.2 Required steps to use the component

The usage of the component is quite clear in itself.
1) Create validators with static factory methods.

- OR
Extend a custom validator from ObjectValidator or primitive validators.
2) Use PrimitiveValidator class to wrap custom validators if validation of various primitive data types is required.
3) Use TypeValidator class to wrap validators if type-specific validation is required.
4) Use composite validators to construct more complex validators with or without a resource bundle information.
5) Validate the data by calling the valid() method.
6) Get the validation message by calling the getMessage() method.
7) Get the validation messages by calling the getMessages() method.
8) Get all the validation messages by calling the getAllMessages() method.

4.3 Demo

The only visible change here for version 3.2, is in the auditing of the CRUD operations.

4.3.1 Validate Strings with length from 6 to 20 characters.

```
StringValidator validator = new StringValidator(IntegerValidator.inRange(6, 20));
// validate and get the error message if the validation failed
String message = validator.valid("Come and get them!");
if(message == null){
        System.out.println("valid");
}else{
        System.out.println("not valid. Cause =  " + message);
}
```

4.3.2 Complex validation

To validate any of the following:

1) Boolean that is false
2) String that contains "fake" as substring
3) Character that is not letter or digit

*These should not take conversion into consideration.

```
// create a aggregating or validator
OrValidator validator = new OrValidator();
// add the first sub validator true if we have a boolean false
validator.addValidator(new TypeValidator(new BooleanValidator(false)
                                          , Boolean.class));
// add the second sub validator, true if we have fake as substring
validator.addValidator(StringValidator.containsSubstring("fake"));

// add a third sub validator, true if a character is NOT alphanumeric
validator.addValidator(new TypeValidator(new NotValidator(
       CharacterValidator.isAlphaNumeric()), Character.class));

validator.valid("this is not fake");         // true, validation success
validator.valid("this is not");              // false, validation failure
validator.valid(new Boolean("false"));       // true, validation success
validator.valid(new Boolean("true"));        // false, validation failure
validator.valid("!");                        // true, validation success
validator.valid("A");                        // false, validation failure
```

4.3.3 *Example of custom validator*

To validate any long or Long that can be converted into double or Strings can be parsed into a long that is multiple of 5.

```
ObjectValidator validator = new LongValidator() {
       public boolean valid(long value) {
       return value % 5 == 0;
       }
       public String getMessage(long value) {
       if (!valid(value)) {
       return "not multiple of 5";
              }
              return null;
       }
};
```

4.3.4 *Example of using a validator to obtain a number of messages from the validation process*

Here we will present the API that is used for getting multiple messages about a single object. Let us assume that the object is a string that represents an email address entered at a website. What we need is the following composite validator:

1) Cannot be null
2) must be between 7 and 35 characters long
3) must end with a ".com"
4) must contain a substring of "@"

We will also assume the following resource bundle setup:

Locale	Key	Message
"En"	"email.field.can.not.be.null"	"Please ensure that the email field is filled in"
"En"	"email.field.is incorrect.length"	"Please ensure that the text for email field is between 7 and 35 characters long – inclusive"
"En"	"email.field.must.end.with .com"	"Please ensure that your email address ends with a .com"
"En"	"email.field.must.contain.the.at.char "	"An email address must contain the @ character."

We will also assume that this set up is in a named bundle: "property"

```
// Create a bundle info to be used
BundleInfo bundleInfo = new BundleInfo();
bundleInfo.setBundle("property", Locale.ENGLISH);
```

```
bundleInfo.setMessageKey("email.field.can.not.be.null");
bundleInfo.setDefaultMessage("email field cannot be null");

// Create a aggregating or validator
AndValidator validator = new AndValidator();

// Add the first sub validator, true if we have a not null string
validator.addValidator(new NotValidator(new NullValidator(bundleInfo)));

bundleInfo.setMessageKey("email.field.is.incorrect.length");
bundleInfo.setDefaultMessage("email field is of the wrong length");

// Add the second sub validator, true if the string is in range of 7..35 chars long
validator.addValidator(StringValidator.hasLength(IntegerValidator.inRange(7,    35,
  bundleInfo)));

bundleInfo.setMessageKey("email.field.must.end.with.com");
bundleInfo.setDefaultMessage("email field doesn't end in com");

// Add the third sub validator, true if the input string end with ".com"
validator.addValidator(StringValidator.endsWith(".com", bundleInfo));

bundleInfo.setMessageKey("email.field.must.contain.the.at.char");
bundleInfo.setDefaultMessage("email field doesn't contain the @ character");

// Add a fourth sub validator, true if the input string contains "@"
ObjectValidator    containsValidator    =    StringValidator.containsSubstring("@",
  bundleInfo);
validator.addValidator(containsValidator);

////////////////////////////////////////////////////////////////////////////////
// Now lets say that we run this validator with the following inputs
// 1) Input string of "hello" and we run the following:
//
// The result will be false, This is because the string is too short
validator.valid("hello");

// This will return "Please ensure that the text for email field is between 7 and 35
  characters long
// – inclusive"
// because the AndValidator will short-circuit on the first failed validator and these are
  done in order in
// which they were placed. So the first validator to fail this is the inRange validator
printResult("validator.getMessage(\"hello\")", validator.getMessage("hello"));

// This will return "Please ensure that the text for email field is between 7 and 35
  characters long
// – inclusive"
// because the AndValidator will short-circuit on the first failed validator and these are
  done in order in
// which they were placed. So the first validator to fail this is the inRange validator and not
  other
// validators will be dealt with. This is not very useful.
printResult("validator.getMessage(\"hello\")", validator.getMessages("hello"));

// This will return the following messages:
//        "Please ensure that the text for email field is between 7 and 35 characters long –
  inclusive"
//        "Please ensure that your email address ends with a .com"
//        "An email address must contain the @ character."
```

```
// Because the AndValidator in this mode will NOT short-circuit on the first failed validator
   and will
// continue doing all validators.
printResult("getAllMessages(\"hello\")", validator.getAllMessages("hello"));

// This will only return the following two messages:
//         "Please ensure that the text for email field is between 7 and 35 characters long –
   inclusive"
//         "Please ensure that your email address ends with a .com"
// Because the AndValidator in this mode will NOT short-circuit on the first failed validator
   but will be
// limited to only 2 messages.
 printResult("getAllMessages(\"hello\", 2)", validator.getAllMessages("hello", 2));

////////////////////////////////////////////////////////////////////////////////
// 2) Here lets assume that we have the string "ivern#topcoder.org" which is invalid and
   lets say that we
// use the non-existent resource bundle of "zh".
//   Assuming the same code as before but with the difference of this line:
validator.removeValidator(containsValidator);
bundleInfo.setBundle("property", Locale.CHINA);
containsValidator = StringValidator.containsSubstring("@", bundleInfo);
validator.addValidator(containsValidator);

// This will return the following messages:
// "email field doesn't end in com"
// "email field doesn't contain the @ character"
// Because the resource bundle was not found and provided default messages have been
   used.
printResult("validator.getAllMessages(\"ivern#topcoder.org\")",
  validator.getAllMessages("ivern#topcoder.org"));
```

5. Future Enhancements

Provide additional validators.