Rounding Factory 1.0 Component Specification

1. Design

The rounding component provides numerous rounding techniques through a standard interface. A strategy pattern is used to load the proper rounding algorithm. Rounding algorithms included in this release are Banker's Rounding, Random rounding and standard rounding. Additional algorithms are easily added to this component by implementing a specific interface.

1.1 Design Patterns

The Singleton pattern is used in the RoundingManager class.
The Strategy pattern is used in the implementation of the RoundingAlgorithm abstract class and its subclasses.
The template method is used the subclass of RoundingAlgorith. Its algorithm relies on an abstract method, round(), which is implemented by subclasses.
The Null pattern is used in the implementation of the RoundingAlgorithm abstract class.

1.2 Industry Standards

None.

1.3 Required Algorithms

1.3.1 Common

The mainly method rounding always has three parameters: the accuracy digit, the compare digit, the string need to be rounded.
In each of the following subsections, let **a** be the accuracy digit, for brevity. In addition, let **b** be the digits immediately following the accuracy digit, and let **c** be the comparison digit.

Rounding up (*down*) means to round such that the value of the number increases (*decreases*).
Any integral value for the accuracy digit is allowed. So, for example, a value of 2 would mean that we would round to **a** precision of hundredths (0.01) and it is not allowed to be negative.

The comparison digit is constrained to be between 1 and 9, inclusive - it is not allowed to be 0 because that would lead to ambiguities.

1.3.2 NoRounding

This algorithm will take the given input and **simply return** it *(If the number contains the sign "+", remove it, and then, return it.)*, ignoring both the accuracy and comparison digits.

1.3.3 DownSymmetricRounding

All numbers move towards zero. Truncate everything after the accuracy digit. If the given number is negative, **the value of number will increase**(the absolute value of the number decreases); if the given number is positive, the number will decrease. If the input is a "round number", return itself with proper precision. That is, if the input given would result from a call to round(), then do nothing except the truncation.
Example:
round(3.21234, 2, x) -> 3.21
round(-2.61234, 2, x) -> -2.61

//

```
//// You can implement the method as follow, ofcourse, you can implement in you own
////way.

// Check the given number, accuracyDigit, and comprisonDigit first.The comparisonDigit
should in the scope of 1 to 9, and the accuracy should be non-negative.
// the given string does not contains the point.
if (number.indexOf(".")!= -1) {                                    // start position 1.
// rounding the number according to the accuracyDigit.
if (acc == 0) {
return number;
} else {

StringBuffer sb = new StringBuffer(number);
sb.append(".")
for (int i = 0; i < acc; ++i) {
sb.append("0");

}
// if the given number is negative, add the sign "-".
return sb.toString();
}                                                                  //end position 1
// the given string contains the point.
} else {
// if the accuracyDigit equals to zero
if (acc == 0) {                                                    //start position 2
if (number.startWiths(".0")
return "0";
else {
return number.startWiths(".") ? "0"
: number.subString(0, number.indexOf(".");
}                                                                  // end position 2
// if the accuracyDigit does not equals to zero.
} else {
if (number.substring(number.indexOf(".")).length() < acc) {
// Append the "0"
} else {
return number.substring(0, number.indexOf(".") + acc);
}
}
}
```

1.3.4 *UpAsymmetricRounding*

All numbers move to zero Truncate everything after the accuracy digit, if the given number is negative, **the value of it will increase**; if the given number is positive, **the value of it will descrease**(The absolute value increase); if the input is a "round number", return itself with proper precision. That is, if the input given would result from a call to round(), then do nothing except the truncation.
Example:
round(3.21234, 2, x) -> 3.22
round(-2.61234, 2, x) -> -2.62

1.3.5 *UpSymmetricRounding*

Truncate everything after the accuracy digit, and round a such that **the value of the number increases**(The absolute value decrease); if the input is a "round number", return itself with proper precision.That is, if the input given would result from a call to round(), then do nothing except the truncation.

Example:
round(3.21234, 2, x) -> 3.22
round(-2.61234, 2, x) -> -2.61

1.3.6 DownAsymmetricRounding

All numbers decrease. Truncate everything after the accuracy digit. If the given number is negative, **the value of number will decrease**(the absolute value of the number increases); if the given number is positive, the number will decrease. if the input is a "round number", return itself with proper precision. That is, if the input given would result from a call to round(), then do nothing except the truncation.
Example:
round(3.21234, 2, x) -> 3.21
round(-2.61234, 2, x) -> -2.62

1.3.7 SymmetricRounding

If **b** is greater than **c**, round up. If **b** is less than or equal to **c**, round down.In this section, round down means **the value of number is descreased**, round up means **the value of number is increased.**
Example:
round(1.26, 1, 5) -> 1.3
round(1.25, 1, 5) -> 1.2
round(-1.25, 1, 5) -> -1.3
round(1.249, 1, 5) -> 1.2

1.3.8 AsymmetricRounding

If **b** is greater than or equal to **c**, round up. Otherwise, round down. In this section, round down means **the value of number is descreased**, round up means **the value of number is increased**.The only different with the *SymmetricRounding* is when **c** is equals to **b.**

Example:
round(1.26, 1, 5) -> 1.3
round(1.25, 1, 5) -> 1.3
round(-1.25, 1, 5) -> -1.2
round(1.249, 1, 5) -> 1.2

1.3.9 BankersRounding

If **b** is greater than **c**, round up. If **b** is less than **c**, round down. Otherwise, set **a** to be the nearest even number (0, 2, 4, 6, or 8).In this section, round up means **the absolute value of number increases**. round down means **the absolute value of number decreases.**

Example:
round(1.06, 1, 5) -> 1.1
round(1.04, 1, 5) -> 1.0

round(1.45, 1, 5) -> 1.4
round(1.55, 1, 5) -> 1.6
round(1.85, 1, 5) -> 1.8
round(1.95, 1, 5) -> 2.0

1.3.10 RandomRounding

If **b** is greater than **c**, round up. If **b** is less than **c**, round down. Otherwise, randomly choose whether to round up or down.It is dependent with the random().
In this section, round up means **the absolute value of number increases**. round down means **the absolute value of number decreases.**

Example:
round(1.251, 1, 5) -> 1.3
round(1.25, 1, 5) -> 1.2 or 1.3
round(1.249, 1, 5) -> 1.2

1.3.11 AlternateRounding

If **b** is greater than **c**, round up. If **b** is less than **c**, round down. Otherwise, alternately round up and down, starting initially with up. This algorithm maintains its state via a private boolean value. In the first time, if it is round up, in the next time, it will be round down.
In this section, round up means **the absolute value of number increases**. round down means **the absolute value of number decreases.**
Example:
round(1.251, 1, 5) -> 1.3
round(1.25, 1, 5) -> 1.3
round(1.25, 1, 5) -> 1.2(again)
round(1.249, 1, 5) -> 1.2

Note:
1. This component does not support scientific notion.
2. If the returned number is non-negative,the developer need not add "+" before it in spite of the given number contains "+" or not.
3. Trailing zeros should be added to pad the number to the correct accuracy, EXCEPT for NoRounding. For example: round(1.2, 3, 2), you should return 1.200 not just 1.2 .BUT, if the return number is 0, you will return itself.
4. If the given number is Infinity, throw exception.
5. If the input is a "round number", return itself with proper precision.
6. If the given number is .123, it should return 0.123.
7. The empty string means the length of string equals to zero after trimming.

1.4 Component Class Overview

Below is a very short overview of the classes in this component. Please refer to the class diagram's documentation tab for a more complete overview of each class。

RoundingManager:
A centralized manager that holds RoundingAlgorithms. The manager implements the Singleton design pattern – there is no need to have more than one around because it can hold every RoundingAlgorithm that you might want.

RoundingAlgorithm:
An abstract class that represents an algorithm to perform rounding. Keeps default accuracy and comparison digit settings. An accuracy digit of a means that we will be rounding to a precision of 10^{-a}. The comparison digit must be between 1 and 9, inclusive.

NoRounding (extends RoundingAlgorithm):
A concrete subclass of RoundingAlgorithm which performs the No Rounding algorithm

UpSymmetricRounding(extends RoundingAlgorithm):
A concrete subclass of RoundingAlgorithm which performs the Always Round Up (Symmetric) rounding algorithm

UpAsymmetricRounding(extends RoundingAlgorithm):

A concrete subclass of RoundingAlgorithm which performs the Always Round Up (Asymmetric) rounding algorithm

DownSymmetricRounding(extends RoundingAlgorithm):
A concrete subclass of RoundingAlgorithm which performs the Always Round Down (Symmetric) rounding algorithm

DownAsymmetricRounding(extends RoundingAlgorithm):
A concrete subclass of RoundingAlgorithm which performs the Always Round Down (Asymmetric) rounding algorithm

SymmetricRounding(extends RoundingAlgorithm):
A concrete subclass of RoundingAlgorithm which performs the Symmetric rounding algorithm

AsymmetricRounding(extends RoundingAlgorithm):
A concrete subclass of RoundingAlgorithm which performs the Asymmetric rounding algorithm

BankersRounding(extends RoundingAlgorithm):
A concrete subclass of RoundingAlgorithm which performs the Banker's rounding algorithm

RandomRounding(extends RoundingAlgorithm):
A concrete subclass of RoundingAlgorithm which performs the Random rounding algorithm.

AlternateRounding(extends RoundingAlgorithm):
A concrete subclass of RoundingAlgorithm which performs the Alternate rounding algorithm

The above concrete subclass of RoundingAlgorithm that implements the appropriate algorithm. See the corresponding subsections of Section 1.3 for more details on each algorithm.

1.5 Component Exception Definitions

ConfigurationException:
This exception is thrown as a wrapper around anything that goes wrong during configuration. Configuration happens only when the private constructor of RoundingManager is called.

RoundingException:
This exception is thrown as a wrapper around anything that can go wrong in any RoundingAlgorithm. This includes but is not limited to: passing an invalid value to the RoundingAlgorithm constructor or calling Round with a String that is not a floating-point number.

NullPointerException:
This exception is thrown by some methods of all classes when an argument to a method is null.

IllegalArgumentException:
Signals that an illegal argument has been passed.

This exception is thrown by different methods when an illegal argument is passed.

1.6 Thread Safety

The component should be thread-safe. First of all, in the Singleton, the developer can lock the class itself. Secondly, Some methods share the same value, the developer should make sure the relevant method is safe.

The relevant methods in RoundingManager are addAlgorithm, getAlgorithm, removeAlgorithm, and clearAlgorithms, you can use the keyword “synchronized” .And in the RoundingAlgorithm, will use an internal lock or other (depending on the developer implementation) to prevent multiple thread access to the RoundingAlgorithm.

2. Environment Requirements

2.1 Environment

- Development language: Java1.4
- Compile target: Java1.3, Java1.4TopCoder Software Components

Note: If the developer want to use the regular expression, you should documented here that this component cannot used in java1.3.

2.2 Third Party Components

- **Configuration Manager v2.1.3.**

  Software applications typically manage application level configuration details in “properties” or “ini” files. In the case of large applications or applications composed of distinct software components, there may be numerous configuration files each bound to a particular functional component. The purpose of the Configuration Manager is to centralize the management of and access to these files.
  The Configuration Manager is used by the RoundingManager to load its configuration from a file.

- **BaseException v1.0**

  In order to handle exceptions and errors in a unified manner, a generic exception class is needed. Error processing logic is simplified since an application can throw one Throwable object wrapping several error types. The Base Exception component follows the chained exception paradigm.
  The ConfigurationException and RoundingException extend the BaseException from the Base Exception component. It has a constructor that receives an exception, so the ConfigurationException is capable of wrapping different configuration exception that may arise, and the RoundingException wrap any exception occurs in rounding, thus not exposing the user to multiple exceptions. The user can still obtain the actual exception that was thrown if he wants to debug.

3. Installation and Configuration

3.1 Package Name

com.topcoder.math.roundingfactory

3.2 Configuration Parameters

```
<?xml version="1.0" encoding="UTF-8" ?>
<ConfigManager>
    <namespace name="com.topcoder.math.roundingfactory">

        <property name="rounding_algorithms">
            <value>no</value>
            <value>random</value>
        </property>

        <property name="no_class">

<value>com.topcoder.math.roundingfactory.Algorithms.NoRounding</value>
```

```
            </property>

            <property name="no_accuracy">
                <value>2</value>
            </property>

            <property name="no_comparison">
                <value>5</value>
            </property>

            <property name="random_class">
                <value>
                com.topcoder.math.roundingfactory.Algorithms.RandomRounding
                </value>
            </property>

            <property name="random_accuracy">
                <value>6</value>
            </property>

            <property name="random_comparison">
                <value>8</value>
            </property>

        </namespace>
    </ConfigManager>
```

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

Extract the component distribution.

4.3 Demo

4.3.1 manipulate

```
// begin by getting an instance of the RoundingManager
RoundingManager rm = RoundingManager.getInstance();

//we can add, get, remove, and clear algorithms
rm.addAlgorithm("nil", new NoRounding());
RoundingAlgorithm algo = rm.getAlgorithm("nil");
algo.setAccuracyDigit(3)
assertEquals(3, algo.getAccuracyDigit());
assertEquals(5, algo.getComparisonDigit());

String roundedValue = algo.round("2.57");
rm.removeAlgorithm("nil");
rm.clearAlgorithms();
```

4.3.2 rounding Algorithm

```
////////////////////////////////////////////////
// create a NoRounding instance
```

```
RoundingAlgorithm algorithm = new NoRounding();

// the number should not be rounded
assertEquals("00123.45600", algorithm.round("+00123.45600", 3, 5));

////////////////////////////////////////////////
// create an UpSymmetricRounding instance
algorithm = new UpSymmetricRounding();

// all numbers move away from zero

assertEquals("1.5", algorithm.round("1.451", 1, 5));
assertEquals("-1.4", algorithm.round("-1.451", 1, 5));

////////////////////////////////////////////////
// create an UpAsymmetricRounding instance
algorithm = new UpAsymmetricRounding();

// all numbers increase
assertEquals("1.5", algorithm.round("1.450", 1, 5));
assertEquals("1.5", algorithm.round("1.451", 1, 5));
assertEquals("-1.5", algorithm.round("-1.450", 1, 5));
assertEquals("-1.5", algorithm.round("-1.451", 1, 5));

////////////////////////////////////////////////
// create a DownSymmetricRounding instance
algorithm = new DownSymmetricRounding();

// all numbers move towards zero
assertEquals("1.4", algorithm.round("1.450", 1, 5));
assertEquals("1.4", algorithm.round("1.451", 1, 5));
assertEquals("-1.4", algorithm.round("-1.450", 1, 5));
assertEquals("-1.4", algorithm.round("-1.451", 1, 5));


////////////////////////////////////////////////
// create a DownAsymmetricRounding instance
algorithm = new DownAsymmetricRounding();

// all numbers decrease
assertEquals("1.4", algorithm.round("1.451", 1, 5));
assertEquals("-1.4", algorithm.round("-1.400", 1, 5));
assertEquals("-1.5", algorithm.round("-1.451", 1, 5));

////////////////////////////////////////////////
// create a SymmetricRounding instance
algorithm = new SymmetricRounding();

// round down if comparison digit equals to truncation
assertEquals("1.4", algorithm.round("1.400", 1, 5));
assertEquals("1.4", algorithm.round("1.450", 1, 5));
assertEquals("1.4", algorithm.round("1.449", 1, 5));
assertEquals("1.5", algorithm.round("1.46", 1, 5));

assertEquals("-1.4", algorithm.round("-1.400", 1, 5));
assertEquals("-1.5", algorithm.round("-1.449", 1, 5));
assertEquals("-1.5", algorithm.round("-1.450", 1, 5));
assertEquals("-1.4", algorithm.round("-1.46", 1, 5));

////////////////////////////////////////////////
// create an AsymmetricRounding instance
algorithm = new AsymmetricRounding();
```

```
// round up if comparison digit equals to truncation
assertEquals("1.4", algorithm.round("1.400", 1, 5));
assertEquals("1.5", algorithm.round("1.450", 1, 5));
assertEquals("1.5", algorithm.round("1.451", 1, 5));
assertEquals("-1.4", algorithm.round("-1.400", 1, 5));
assertEquals("-1.4", algorithm.round("-1.450", 1, 5));
assertEquals("-1.5", algorithm.round("-1.451", 1, 5));

////////////////////////////////////////////////////
// create a BankersRounding instance
algorithm = new BankersRounding();

// round to nearest even number
assertEquals("1.0", algorithm.round("1.05", 1, 5));
assertEquals("1.2", algorithm.round("1.15", 1, 5));
assertEquals("1.4", algorithm.round("1.45", 1, 5));
assertEquals("-1.6", algorithm.round("-1.55", 1, 5));
assertEquals("-1.8", algorithm.round("-1.85", 1, 5));
assertEquals("-2.0", algorithm.round("-1.95", 1, 5));


////////////////////////////////////////////////////
// create a RandomRounding instance
algorithm = new RandomRounding();

// round randomly
string result =  algorithm.round("1.450", 1, 5);
assertTrue(result == "1.4" || result == "1.5");
result =  algorithm.round("-1.450", 1, 5);
Assert.IsTrue(result == "-1.4" || result == "-1.5");


////////////////////////////////////////////////////
// create a AlternateRounding instance
algorithm = new AlternateRounding();

// round alternately
assertEquals("1.4", algorithm.round("1.450", 1, 5));
assertEquals("-1.5", algorithm.round("-1.450", 1, 5));
```

4.4 Future Enhancements

The most important thing is that the Strategy design pattern makes it easy to add new rounding algorithms, which is the most obvious extension of this component.

Now, the component does not support the scientific notion, it should be thought of for future enhancement.