Configuration API 1.1 Component Specification

1. Design

All changes performed when synchronizing documentation with the version 1.0 of the source code of this component and fixed errors in the CS are marked with **purple**.

All changes made in the version 1.1 are marked with **blue**.

All new items in the version 1.1 are marked with **red**.

In a new paradigm the configuration model will be implemented by a separate component from the configuration manager component, which will be primarily concerned with file system operations and other implementation details. This component will define the API for a configuration interface that components can rely on without being coupled to the Configuration Manager component.

A ConfigurationObject is an object that contains configuration information. It contains some properties, which map a key to a list of values. And it also contains zero or more nested ConfigurationObject instances. Many methods are provided to manipulate the properties and nested configuration objects. And especially, some methods utilize regular expression of wildcard path to find descendant configuration objects, and also operate on these matched descendants.

BaseConfigurationObject is provided as a default adapter for custom implementations of ConfigurationObject. It implements all the methods by throwing exception. And TemplateConfigurationObject is defined to provide a default implementation of methods related to descendants. It simply utilizes other methods in the interface.

A default implementation of ConfigurationObject is defined as DefaultConfigurationObject. It holds all the properties and ConfigurationObject in memory. It also implements Cloneable interface to provide a deep clone operation, and implements Serializable interface.

SynchronizedConfigurationObject is used to wrap any ConfigurationObject implementation. It provides an easy way to synchronize any ConfigurationObject instance.

To support arbitrary operation on descendants, Process interface is defined. Instances of this interface can be passed to ConfigurationObject to process wildcard matched descendants.

Changes in the version 1.1:

- Development language is changed to Java 1.5. Thus generic parameters are explicitly specified for all generic types in the source code.
- Added ConfigurationObject#getPropertyValue() and ConfigurationObject#getPropertyValues() overloads with “required” parameter. They throw an exception if required property is missing.
- Added support of generic return type for ConfigurationObject#getPropertyValue() and ConfigurationObject#getPropertyValues() methods.
- Added ConfigurationObject#getXXXProperty() methods that additionally can perform parsing of Integer, Long, Double, Date and Class instances from String property values.
- Using Base Exception 2.0 as a dependency (instead of 1.0).
- Some trivial source code changes are performed to make the component meet TopCoder standards.

1.1 *Design Patterns*

Template Method

The TemplateConfigurationObject uses template methods. Operations on specific descendants depend on operation to find the descendants and operations on a single ConfigurationObject.

Decorator

The synchronized wrapper acts as a thread-safe decorator for configuration object implementation, attaching thread safety to its target dynamically.

Composite

This design utilizes composite pattern in a great deal, because this design treats a single value configuration and a collection of sub-configurations identically.

Strategy

Processor instance passed to ConfigurationObject#processDescendants is a strategy. ConfigurationObject uses this strategy to process all the matched descendants.

1.2 Industry Standards

None

1.3 Required Algorithms

1.3.1 Get descendants of a ConfigurationObject.

There are two overloads of getting descendants methods. One is getting all of the descendants, and the other is getting the descendants whose names can be matched to some given pattern. These two processes would look much the same. We can use BFS to complete this task. Here is the pseudo code.

```
// create a set to hold all the visited descendants.
Set visited = new HashSet();

// create a list to mimic queue used in bfs, using LinkedList as
//a Queue
List queue = new LinkedList();

// initialize the queue
visited.add(this);
queue.add(this);

// breadth first search
while (queue.size() > 0) {
    ConfigurationObject obj = queue.removeFirst();

    foreach (ConfigurationObject child in obj.getAllChildren()) {
        if (!visited.contains(child)) {
            visited.add(child);
            queue.addLast(child);
        }
    }
}

// if a filtering pattern is given,
// remove all the un-matched object in the queue
//if self no considered as descendants
visited.remove(this);
return queue.toArray();
```

1.3.2 *Find descendants by wildcard.*

The process can be divided into two parts. The outer part is to split the given path into several layers, and to match the layers one by one. The inner part is doing wildcard match.

Following is a brief implementation of the outer parts.

1) Trim whitespaces, '/' and '\\' characters from both ends.

```
path = trimSlashes(path.trim());
// trimSlashed should be implemented somewhere.
```

2) Split path to a string array.

```
String[] paths = split("[\\\\/]");
```

3) Match the path layer by layer

```
// create a map to hold child-parent relationship
// it is only used for delete method.

// initialize first layer
Set current = new HashSet();
Set next = new HashSet ();
current.add(this);

// match each sub-path
foreach (String subPath in paths) {
    // create a list to hold next layer

    foreach(ConfigurationObject obj in current) {
        foreach (ConfigurationObject child in obj.getAllChilds())
{

            // match the name of current layer
            if (wildMatch(subPath, child.getName()) {
                next.add(child);
               //if it is the delete action, delete the last layer children
                if (subPath == paths[paths.length - 1] && isDelete){
                        obj.removeChild(child.getName());
                }
            }
        }
  }
   current.clear();
   current.addAll(next);
   next.clear();
 }
return current;
```

Following is an implementation of wildcard match algorithm, using dynamic programming strategy:

```
boolean wildMatch(String pattern, String str) {
    boolean[][] status = new boolean[pattern.length() +
1][str.length() +
            1];

        for (int i = 0; i <= pattern.length(); i++) {
            for (int j = 0; j <= str.length(); j++) {
                status[i][j] = false;
            }
        }

        status[pattern.length()][str.length()] = true;

        //the follow steps are using dynamic programming,
        //status[i][j] == true means pattern.subString(i) matches
str.subString(j)
        for (int i = pattern.length() - 1; i >= 0; i--) {
            for (int j = str.length() - 1; j >= 0; j--) {
                char p = pattern.charAt(i);
                char s = str.charAt(j);

                //'*' can be used to match any letters
                if (p == '*') {
                    status[i][j] |= status[i + 1][j + 1];
                    status[i][j] |= status[i + 1][j];
                    status[i][j] |= status[i][j + 1];
                } else if ((p == '?') || (p == s)) {
                    status[i][j] |= status[i + 1][j + 1];
                }
            }
        }

        //return whether the whole String can match
        return status[0][0];
}
```

1.3.3 *Clone implementation.*

Because the implementation of ConfigurationObject graph is not a tree, it is only a directed acyclic graph. In order to keep the DAG structure when cloning, some cache mechanisms should be provided. In other words, we don't want the same object to be cloned more than once.

To solve this problem, we add a private clone method, which takes a Map parameter. The map is a cache, which maps the original objects to their clones. Here is a sample implementation.

```
      DefaultConfigurationObject clone = (DefaultConfigurationObject)
super.clone();

      // clone the properties
      clone.properties = new HashMap(this.properties);

      // iterate every child to clone it.
      for (Iterator itr = children.values().iterator(); itr.hasNext(); )
      {
          DefaultConfigurationObject child =
       (DefaultConfigurationObject) itr.next();

          // if the child is not cloned yet, clone it,
          // and put it into the cache.
```

```
    if (!cache.containsKey(child)) {
        cache.put(child, child.clone(cache));
    }

    // retrieve the child's clone from cache
    clone.children.put(child.getName(), cache.get(child));
}
return clone;
```

At last, in the public clone method, we can simply call this.clone(new HashMap());

1.4 *Component Class Overview*

ConfigurationObject [interface]

A ConfigurationObject is an object which contains configuration information.

ConfigurationObject can have zero or more properties associated with it. All the properties are consisted of a String key and a list of values. Property key must be unique in the same ConfigurationObject.

ConfigurationObject can also contain zero or more child ConfigurationObject. The children are uniquely identified by their names. There is no restriction on the child-parent relationships in this interface (API definition). Some implementations may only allow tree structure, or only allow DAG, and so on.

Methods in this interface can be categorized in two dimensions. One dimension divides methods into two categories: properties operations, and children operations. The other dimension divides methods by the way to search properties and children. The direct way to search properties and children is to use exactly their names. The second way is to use regular expression to match properties or children names. And the third way is to use wildcard match (like in UNIX file system) to find children.

Changes in 1.1:

- getPropertyValue() method was updated to use generic parameter for return value casting.

- Added getPropertyValues() overload with generic parameter that is expected to return array of specific type.

- Added getPropertyValue() and getPropertyValues() overloads that accept "required" parameter and throw exception if required property is missing.

- Added getXXXProperty() methods that additionally can perform parsing of Integer, Long, Double, Date and Class instances from String property values.

BaseConfigurationObject

An abstract adapter implementation of ConfigurationObject interface. All the methods in this class always throw UnsupportedOperationException. This class exists as convenience for creating custom ConfigurationObject. Extend this class to create a custom ConfigurationObject and override the only the methods which can be supported by certain configuration strategy.

This class has no state, and thus it is thread safe.

Changes in 1.1:

- getPropertyValue() method was updated to use generic parameter for return value casting.

- Added getPropertyValues() overload with generic parameter that is expected to return array of specific type.

- Added getPropertyValue() and getPropertyValues() overloads that accept "required" parameter and throw exception if required property is missing.

- getPropertyValue() and getPropertyValues() methods without "required" parameter were updated to delegate execution to overloads using required=false parameter value.

- Added getXXXProperty() methods that additionally can perform parsing of Integer, Long, Double, Date and Class instance from String property values.

TemplateConfigurationObject

This class uses Template Method design pattern to implement some methods in ConfigurationObject interface.

In ConfigurationObject interface, many methods operate on some descendant ConfigurationObject which are found by path containing wildcard. Because the operation on a single ConfigurationObject is also defined, we just first use "findDescendants" to find all the matched descendants, and then invoke corresponding simple method on them. In this case, "findDescendants" method and other simple methods are template methods.

This class itself contains no state, and thread safe depends on whether template methods are thread-safe.

DefaultConfigurationObject

Default implementation of ConfigurationObject. It extends from TemplateConfigurationObject to utilize the implemented methods in it.

This class uses a Map in memory to hold properties. The key of Map is a String representing the property key. The value of Map is a List instance containing all the property values (null is allowed). And also a Map in memory is used to hold child ConfigurationObjects. The key of this Map is a String representing the name of child object. And the value is the child instance. The relationship graph of this implementation should always be a DAG.

Besides ConfigurationObject interface, this class also implements Serializable and Cloneable interface. To support Serializable interface, it just ensures that all the coming properties values and child objects are instances of Serializable.

To support Cloneable interface, because we want to keep DAG structure, a clone overload with a cache parameter is provided. In this case, all the children should be instance of DefaultConfigurationObject.

This class is mutable and not thread safe.

Changes in 1.1:

- Specified generic parameters for all generic types.

- getPropertyValue() and getPropertyValues() methods were updated to use generic parameters for return value casting. "required" parameters were added to these methods.

- Added getXXXProperty() methods that additionally can perform parsing of Integer, Long, Double, Date and Class instance from String property values.

SynchronizedConfigurationObject

It is a synchronized wrapper of any ConfigurationObject.

Every method call should be synchronized on the inner ConfigurationObject before delegating to the inner ConfigurationObject. But please note that, this wrapper can only ensure the methods declared in ConfigurationObject work together thread safely, because only these methods are synchronized. And extension of this class should lock the inner ConfigurationObject to ensure thread safe.

The inner ConfigurationObject can be accessed by protected getter to let subclasses to use them.

This class is thread-safe. All the methods are synchronized on the same object.

Changes in 1.1:

- getPropertyValue() method was updated to use generic parameter for return value casting.

- Added getPropertyValues() overload with generic parameter that is expected to return array of specific type.

- Added getPropertyValue() and getPropertyValues() overloads that accept "required" parameter and throw exception if required property is missing.

- Added getXXXProperty() methods that additionally can perform parsing of Integer, Long, Double, Date and Class instance from String property values.

Processor [interface]

This interface defines the contract for ConfigurationObject processor. It only contains a method named as process, which takes a ConfigurationObject parameter and returns nothing.

Typically, it would be passed to ConfigurationObject#processDescendants to process all the descendants.

1.5 *Component Exception Definitions*

ConfigurationException

This class is the base exception of this component. It provides user the supports for dealing with all the exceptions from this component as a whole.

Currently, there are two extensions of this exception, ConfigurationAccessException and InvalidConfigurationException.
This class is not thread safe because its base class is not thread safe.
Changes in 1.1:
- Extends BaseCriticalException instead of BaseException
- Added new constructors to meet TopCoder standards

InvalidConfigurationException

This exception indicates given property key, property value, child name or child instance is not acceptable by specific ConfigurationObject implementation, like a cycle occurs after adding some child. For example, the default implementation requires all the property values are instances of Serialize able.
It can be thrown from methods updating properties and adding children. In the default implementation, this exception is thrown for non-Serialize able property values, etc.
This class is not thread safe because its base class is not thread safe.
Changes in 1.1:
- Added new constructors to meet TopCoder standards

ConfigurationAccessException

This exception indicates an error occurs while accessing (reading or writing) the configuration. It may cause by IO problem, database connection problem, or etc.
It can be thrown from almost all of methods of ConfigurationObject interface. But in the default implementation, this exception is never thrown, because default implementation is in memory.
This class is not thread safe because its base class is not thread safe.
Changes in 1.1:
- Added new constructors to meet TopCoder standards

ProcessException

This exception indicates an error occurs while processing ConfigurationObject instances. It can be thrown from Processor implementations, and ConfigurationObject#processDescendants method.
This class is not thread safe because its base class is not thread safe.
Changes in 1.1:
- Added new constructors to meet TopCoder standards

PropertyTypeMismatchException

This exception is thrown by implementations of ConfigurationObject when value cannot be casted to expected type or parsed from string.
This class is not thread safe because its base class is not thread safe.

PropertyNotFoundException

This exception is thrown by implementations of ConfigurationObject when some required property is missing.

This class is not thread safe because its base class is not thread safe.

1.6 *Thread Safety*

The default implementation of ConfigurationObject is not thread safe, because it is mutable. But a synchronized wrapper is provided for both ConfigurationObject interface and DefaultConfigurationObject. The synchronized wrappers provide a simple way to add thread safe property to any implementation of ConfigurationObject interface.

SynchronizedConfigurationObject locks the inner ConfigurationObject to synchronize all the method calls. When being called, it first tries to lock the inner ConfigurationObject, and then delegate the call wrapped ConfigurationObject instance. Extensions of this class should also locks the inner object to ensure thread safety.

Thread safety of this component was not changed in the version 1.1.

2. Environment Requirements

2.1 *Environment*

Development language: Java 1.5
Compile target: Java 1.5, Java 1.6
QA Environment: Solaris 7, RedHat Linux 7.1, Windows 2000, Windows 2003

2.2 *TopCoder Software Components*

Base Exception 2.0 – defines a base class for custom exceptions.

NOTE: The default location for TopCoder Software component jars is../lib/tcs/COMPONENT_NAME/COMPONENT_VERSION relative to the component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location.

2.3 *Third Party Components*

None

NOTE: The default location for 3rd party packages is ../lib relative to this component installation. Setting the ext_libdir property in topcoder_global.properties will overwrite this default location.

3. Installation and Configuration

3.1 *Package Name*

com.topcoder.configuration

com.topcoder.configuration.defaults

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

See demo below.

4.3 Demo

4.3.1 Create ConfigurationObject

```
 // create a DefaultConfigurationObject
ConfigurationObject defaultCo = new DefaultConfigurationObject("the name");
// create a SynchronizedConfigurationObject with the inner object
ConfigurationObject synchronizedCo = new SynchronizedConfigurationObject(defaultCo);
// DefaultConfigurationObject is also can be used as TemplateConfigurationObject
TemplateConfigurationObject templateCo = (TemplateConfigurationObject) defaultCo;
```

4.3.2 Manipulate ConfigurationObject properties

```
// set the value, can be null, and the old value will be returned
Object[] values = defaultCo.setPropertyValue("key", "value");
// set a array of values with the key
values = defaultCo.setPropertyValues("key", new Object[] {"value1", "value2"});

// check whether a ConfigurationObject contains a key
boolean contained = defaultCo.containsProperty("key");

// get all the values with the key
values = defaultCo.getPropertyValues("key");
// get the first value of the key
Object value = defaultCo.getPropertyValue("key" , Object.class);
// get the count of values with the key
int count = defaultCo.getPropertyValuesCount("key");

// remove the values of the key
defaultCo.removeProperty("key");
defaultCo.clearChildren();
// get all the keys of properties
String[] keys = defaultCo.getAllPropertyKeys();
// get the keys with the regex pattern
 keys = defaultCo.getPropertyKeys("[a\\*b]");
```

4.3.3 Manipulate nested ConfigurationObject

```
// add a child
DefaultConfigurationObject child = new DefaultConfigurationObject("child");
defaultCo.addChild(child);

// check contains child
boolean contained = defaultCo.containsChild("child");
// get the child by name
ConfigurationObject thechild = defaultCo.getChild("child");
// remove the child by name
thechild = defaultCo.removeChild("child");
// clear the child
defaultCo.clearChildren();

// get all the children
ConfigurationObject[] children = defaultCo.getAllChildren();
// get all the children by a regex pattern
 children = defaultCo.getChildren("[abc]");
```

4.3.4 Manipulate descendants by key aggregation

```
 // get all the descendants
ConfigurationObject[] descendants = defaultCo.getAllDescendants();
// find the descendants by a path
descendants = defaultCo.findDescendants("path");
// delete the descendants by a path
descendants = defaultCo.deleteDescendants("path");
// get descendants with the regex pattern
descendants = defaultCo.getDescendants("pattern");
```

4.3.5 Use clone and synchronized wrapper.

```
 // clone the ConfigurationObject
ConfigurationObject clone = (ConfigurationObject) defaultCo.clone();
ConfigurationObject synchronizedCo = new SynchronizedConfigurationObject(clone);
 SynchronizedConfigurationObject synchronizedClone = (SynchronizedConfigurationObject) synchronizedCo.clone();
```

4.3.6 Use as TemplateConfigurationObject

```
 DefaultConfigurationObject child = new DefaultConfigurationObject("child");
TemplateConfigurationObject templateCo = (TemplateConfigurationObject) defaultCo;
// set the property value with a path
templateCo.setPropertyValue("a", "key", "value");

// set the property values with a path
templateCo.setPropertyValues("a/b", "key", new Object[] {"value"});

// remove the property values with a path
templateCo.removeProperty("a*\\/b", "key");

// clear property with a path
templateCo.clearProperties("path/*c");
// add a child with a path
templateCo.addChild("path", child);
// remove child with a path and child name
templateCo.removeChild("path", child.getName());
// clear children with a path
templateCo.clearChildren("b");

// processDescendants with a path
templateCo.processDescendants("path", new ProcessorMock());
```

4.3.7 Retrieval of property values with casting and parsing

```
 // Create an instance of DefaultConfigurationObject
ConfigurationObject config = new DefaultConfigurationObject("default");

// Initialize sample properties
config.setPropertyValues("ints", new Object[] {1, 2, 3});
config.setPropertyValues("strings", new Object[] {"abc", "def"});
config.setPropertyValue("intValue1", 5);
config.setPropertyValue("intValue2", "5");
config.setPropertyValue("longValue", "12345");
config.setPropertyValue("doubleValue", "1.23");
Calendar calendar = Calendar.getInstance();
calendar.clear();
calendar.set(2011, 0 1);
config.setPropertyValue("dateValue1", calendar.getTime());
config.setPropertyValue("dateValue2", "2011-01-01");
config.setPropertyValue("class", "java.lang.Integer");

// Retrieve the property values as integer array
Integer[] intValues = config.getPropertyValues("ints", Integer.class);
// intValues must contain {1, 2, 3}

// Retrieve the property values as string array
String[] stringValues = config.getPropertyValues("strings", String.class);
// stringValues must contain {"abc", "def"}

// Retrieve the integer property value (without parsing support)
Integer intValue = config.getPropertyValue("intValue1" , Integer.class);
// intValue must be equal to 5

// Retrieve the integer property value by parsing it from string
intValue = config.getIntegerProperty("intValue2", true);
// intValue must be equal to 5

// Retrieve the long property value by parsing it from string
Long longValue = config.getLongProperty("longValue", true);
// longValue must be equal to 12345
// Retrieve the double property value by parsing it from string
Double doubleValue = config.getDoubleProperty("doubleValue", true);
// doubleValue must be equal to 1.23

// Retrieve the date property stored as Date
Date dateValue1 = config.getDateProperty("dateValue1", "yyyy-MM-dd", true);

// Retrieve the date property stored as String
Date dateValue2 = config.getDateProperty("dateValue2", "yyyy-MM-dd", true);

// dateValue1.getTime() must be equal to dateValue2.getTime()
// Both dates must represent 2011-01-01

// Retrieve the class property value
Class<?> clazz = config.getClassProperty("class", true);
 // clazz must be equal to Integer.class
```

4.3.8 Retrieval of required property values

```
 // Create an instance of DefaultConfigurationObject
ConfigurationObject config = new DefaultConfigurationObject("default");

// Retrieve optional not existing property
Object value = config.getPropertyValue("key1" , Object.class, false);
// value must be equal to null

try {
    // Retrieve required, but not existing property
    value = config.getPropertyValue("key2", Object.class, true);
    // PropertyNotFoundException must be thrown here
} catch (PropertyNotFoundException e) {
    // Ignore
 }
```

5. Future Enhancements

Add more implementation of ConfigurationObject API.