Configuration Persistence 1.0 Component Specification

1. Design

This component will provide compatibility with configuration manager persistence files (XML and properties) via the new preferred Configuration API. It is expected that only the application will need to directly access this component and that components used by the application will be configured using instances of the Configuration Object interface defined by the Configuration API component.

1.0.1 Design overview

This design meets the requirements of loading both types of persistence files by default, and offers a strategy that will allow an application to specify other formats of files to use. It is not designed to be easily extended to non-URL persistence mechanisms like a database, but is specialized for loading configuration to and from files. It can load and save configuration files, and can also be used to create new configuration files.

It stores a single root ConfigurationObject named "Configuration" that stores all configuration loaded by the component in descendent ConfigurationObjects. When initialized, it can read configuration files to load either from a file (which can be in any supported format) or from files programmatically specified with either a ConfigurationObject or a Map. It supports both namespaces and nested properties by using nested ConfigurationObjects.

The ConfigurationObject produced by this component will have a directed tree structure having a single root node named "Configuration".

1.1 Design Patterns

Strategy pattern has been used to allow the using application to specify custom file formats. In this pattern, ConfigurationFileManager provides the context, ConfigurationPersistence is the abstract strategy, and PropertyFilePersistence and XMLFilePersistence are the currently provided concrete strategies.

Data Access Object pattern has been used in the persistence implementations, which isolate the ConfigurationFileManager class from the actual persistence used.

1.2 Industry Standards

- Xml

1.3 Required Algorithms

1.3.1 Translation of Properties to Configuration Objects

Configuration manager persistence files support the definition of nested properties, like this:

```
property.propertyA.propertyB = Value
property.propertyC = ValueA
```

This component interpret the nested property structure defined by Configuration manager as a nested ConfigurationObject with the bottom level defining the actual property keys and each higher level defining a child ConfigurationObject. For example, the above named properties read from a file would result in a ConfigurationObject named Property with a child ConfigurationObject named PropertyA and a property with the key PropertyC and value ValueA. The child

ConfigurationObject PropertyA would have a property with the key PropertyB and value Value.

Namespaces from the old Configuration Manager component will be interpreted as a nested ConfigurationObject structure. For example, consider the following properties files:

```
ConfigManager.properties:
       com.topcoder.namespace = file1.properties
file1.properties:
       property.propertyA = value
```

Here, we will have the following ConfigurationObject structure:
"Configuration" has child "com.topcoder.namespace" has child "property" has property "propertyA" with value "value".

The remainder of the algorithms section provides more concrete details on how to implement this logic in various circumstances.

1.3.2 Keeping Track of Files

There will be a map from namespace to file in the configuration manager, we will save all namespaces whose are invoked in the loadFile() and createFile() methods.

1.3.3 Configuring the component

There are two things that need to be specified in configuring the component: the mapping of file type to persistence object, and the namespaces and files to load into configuration.

The mapping of file type to persistence object is completely optional; if nothing is provided, the default persistence objects that load the configuration file formats defined in the Configuration Manager component will be used. If it is provided, the mapping must be provided in a Map or ConfigurationObject; reading the persistence types from a file is not supported because they are needed to read the configuration file. This part of the configuration is processed first. Except for the case where the mapping is provided as part of a ConfigurationObject, this part of the configuration is not covered in detail here because it is accomplished through simple copying of keys and values into an internal Map.

Mapping of namespaces to files can be accomplished using either a file, which may be in any supported format, or programmatically with a Map or ConfigurationObject.

Reading Configuration From A File
A configuration file for this component will contain a map of namespaces in this form "top.sub1.sub2…" to files. Each namespace will be mapped to the relative path to a file.

The relative file path can refer to a file in the classpath or in the application root. Please refer to CS 1.3.11 to find the file. The URL for the specified file will be retained in the 'files' ConfigurationObject.

For namespace/file pair listed in the configuration file, a ConfigurationObject for the specified file will be loaded using the appropriate ConfigurationPersistence object. This object will be determined from the file extension of the specified file. The extension will be compared to the extension for each FileType in the persistenceMap.

When a match is found, the ConfigurationPersistence mapped to that FileType will be used.

The resulting ConfigurationObject will be added as a child to the configuration tree at the node specified by the namespace. For example, if a namespace A.B.C is mapped to a file, the loaded ConfigurationObject will be named C and added as a child to ConfigurationObject B which will be a child of ConfigurationObject A which will be a child of the root ConfigurationObject, named "Configuration".

Reading Configuration From A ConfigurationObject
This component may be configured with a ConfigurationObject. The given ConfigurationObject should have a child ConfigurationObject named "files" and optionally a second child ConfigurationObject named "persistence". The "files" configuration object will contain properties as described under "Reading Configuration From A File" - the keys will be namespaces, and the values file paths and possibly file roots (as strings). The "persistence" ConfigurationObject, which should be processed first if it is present, will have two values for each key: one FileType, and one ConfigurationPersistence. The keys don't particularly matter, except that each is associated with exactly two values. Each ConfigurationPersistence will be associated with the corresponding FileType in the persistenceMap.

Configuration From a Map
The component can be configured with Maps. There are two constructors that take Maps; one takes a single Map specifying the namespaces and files to load, and the second takes two Maps, of which the second is used to configure persistence. This second map will have FileType objects as keys and ConfigurationPersistence objects as values, and simply needs to be copied to the internal persistenceMap. The first map can be processed as described above under "Reading Configuration From A File."

1.3.4 Loading Files After Configuration
Additional files can be loaded into the ConfigurationObject tree after the instantiation of the ConfigurationFileManager object. This requires the user to specify a namespace to load the file into. This works exactly like specifying a namespace during configuration. The namespace is split on ".". The specified file is loaded, and the ConfigurationObject generated is given the name of the last part of the namespace, and placed in a ConfigurationObject specified by the preceding parts of the namespace.

1.3.5 Saving Configuration Changes
The component maintains its internal ConfigurationObject privately - all ConfigurationObjects passed out of the component are copies of the internal object. Any changes an application makes to ConfigurationObjects retrieved from the component will not be reflected in the internal ConfigurationObject, so the internal ConfigurationObject will always reflect the state of the files when they were loaded or last refreshed. When a user wants to update the configuration files through this component, an altered ConfigurationObject will be provided. This object should have the entire configuration tree represented in the ConfigurationFileManager. No files need to be specified, because the ConfigurationFileManager maintains the Map of the various stored ConfigurationObjects to the files they represent.

When an attempt is made to save an altered configuration, these steps should be followed:

1. Compare the passed in ConfigurationObject with the internal ConfigurationObject and identify any differences.
2. For each descendent ConfigurationObject that is changed, identify the associated file and load the current state of the file.

If the file loaded differs from the internal representation, throw a ConfigurationUpdateConflictException - the file has been changed since the last time it was loaded, so the using application has updated an out of date version of the configuration. In addition, update the internal representation of the configuration file to match the current state of the configuration file. This will ensure that there will be no known invalid data presented in the ConfigurationFileManager.

If the file loaded is the same as the internal representation, save the updated configuration to the file using the appropriate ConfigurationPersistence object and update the internal representation to match the updated file.

The comparison in step one should be accomplished through a breadth-first search of the ConfigurationObject tree. If a branch is missing, it will be necessary to identify all of the affected files by searching the corresponding branch of the 'files' ConfigurationObject. In this case, all corresponding properties will simply be removed from the files. If any property in the provided ConfigurationObject is changed, missing, or added in comparison with the currentConfiguration ConfigurationObject, the corresponding file should be saved using the proper ConfigurationPersistence and then the currentConfiguration should be updated to match. New branches that cannot be associated with any particular file are not permitted. This will occur if a branch appears above any "file" property in the files ConfigurationObject. If it appears at the same level or below, it will be assumed that the new branch is a new complex (nested) property.

This method has an optional namespace argument. If this argument is supplied, only the specified sub-tree of the overall ConfigurationObject should be saved. The only difference in this case is that the breadth-first search is started at the node specified by the namespace rather than at the root node.

1.3.6 Accessing Files Safely

File comparison and update needs to be done atomically. No other thread or process may modify a file while this component loads a file, compares it with the previously loaded version, and saves the updated version.

In order to accomplish this, two separate cases must be addressed. It should not be possible for another ConfigurationFileManager instance to access and update the file, and it should not be possible for another application to modify the file while it is being accessed by this component.

The first case is addressed by synchronizing those sections of code that access, check, and update files on the class itself. It is not sufficient to synchronize simply, which will prevent additional access from the same ConfigurationFileManager instance, as the file can still be modified from another ConfigurationFileManager instance.

The second case is addressed through use of a FileLock. This class is in the java.nio package of Java 1.4, and is used through a FileChannel that can be retrieved from a FileStream. Getting an exclusive lock on a file before access will prevent other

processes from modifying or accessing the file while it is being checked and updated. However, it won't prevent other threads in the same application from accessing and modifying the file, so the above synchronization strategy is still required.

1.3.7 Creating New Configuration Files

This component can save ConfigurationObjects to new files, and at the same time add these ConfigurationObjects to the currentConfiguration tree. The createFile() method takes arguments specifying the namespace to add to the currentConfiguration, the file to save the ConfigurationObject to, and the ConfigurationObject to save. If it doesn't already exist, the specified directory should be created. If the given file already exists, it will be overwritten. The actual saving will be performed by the appropriate ConfigurationPersistence. The namespace must not conflict with a namespace already associated with a file. Once the ConfigurationObject is successfully saved, it will be added to the currentConfiguration at the level specified by the namespace argument.

1.3.8 Refreshing Configuration

There is no guarantee that the files from which configuration is loaded will stay the same indefinitely. It may be necessary at some point to refresh the configuration from the current files. The refresh() method will cause ConfigurationFileManager to reload all of the configuration in currently represented files.

This can be accomplished by running a breadth-first search on the files ConfigurationObject. Each time a "file" property is encountered, the corresponding file will be loaded using the proper ConfigurationPersistence object. These loaded files will be added to a new ConfigurationObject representing the updated configuration. Once the search is complete and all files are loaded, the new ConfigurationObject will be copied to the currentConfiguration field.
It is also possible to specify some subset of the overall configuration tree to reload by specifying a namespace to load. If such a namespace is given, only that portion of the files ConfigurationObject under the specified namespace will be searched and the generated ConfigurationObject will be saved over the corresponding branch of currentConfiguration.

1.3.9 Reading and Writing Property Files

The property files in this component have exactly the same format as the property files in the Configuration Manager component, and can be handled in the same way. A working implementation of these operations exists in the PropConfigProperties.java file in that component. One slight difference exists in that we will allow the same property to be defined more than once, in which case all associated values will be kept.
To read a property file, first prepare a reader from the specified file. Read each line from the reader. For each line read, skip it if it's a comment or empty line. If it contains a property, split the line on '=', ':', or ' ' to separate the key from the value list. Try each of those symbols in turn. Once the key and value list are separate, parse the value list into a list of values by splitting on ';' or the list delimiter specified in the file (e.g. ListDelimiter=,). If the key is for a nested property, it will contain '.' - if it does, create a nested ConfigurationObject structure as described above to represent it. The last section of the property will always be stored as a property key, and each preceding section will define a child ConfigurationObject between that property and the ‘default’ ConfigurationObject.

To write a property file, execute a breadth first search through the ‘default’ ConfigurationObject, writing a line for each property key found. The key should include each parent ConfigurationObject (not including the root) separated by a '.', followed by the property key. The value list should contain a string representation of each value associated with that property, separated by ';'.

1.3.10 Reading and Writing Xml Files

The xml files used in this component have exactly the same format as the xml files in Configuration Manager, and can be handled the same way. A working implementation of these operations exists in the XmlConfigProperties.java file in that component. The xml schema file included in this distribution is copied without alteration from that component.

To read an xml file, first parse the file into an org.w3c.xml.Document file using a DocumentBuilder and DocumentBuilderFactory. Starting with the root node, get each child node. Ignore comment nodes, and parse Property nodes. If a property node has child properties, it should be a child ConfigurationObject. If it doesn't, it will be a property key with the associated value. If a property has both, it should be used as both a property key and as a child ConfigurationObject.

To write an xml file, search the ConfigurationObject tree depth first, writing all nested properties for one child ConfigurationObject before proceeding to the next. Any key-value pairs will simply be stored as simple properties.

For each root’ child node, if its is named <Config>, create ConfigurationObject with the given name, otherwise, add properties into the ‘default’ ConfigurationObject.

1.3.11 Locating files

Files will be specified as relative or absolute paths. The component first attempts to resolve them via the context ClassLoader (Thread.currentThread().getContextClassLoader().getResource()), and if that fails then it attempts to locate them by the given name on the file system. Files located via the ClassLoader can be inside archive files; the component can read configuration from such files, but it cannot update them. Once a file is found, the File should be saved in the "files" ConfigurationObject under the appropriate namespace.

File access needs to be done in a safe way, using a FileLock. A FileLock can be acquired on a FileChannel acquired from a RandomAccessFile constructed using a given File. Once the FileLock is acquired, the file can be read in the normal manner for a properties file or for an xml file, since the FileLock will only prevent other applications from accessing the file. This requirement is relaxed for configuration files inside archives or otherwise not residing as independent files on the file system. Such files cannot be freely updated anyway (at least, not by this component), and it is risky to take out an exclusive lock on a whole archive file without knowing its purpose and uses.

All input and output operations should be synchronized against the ConfigurationFileManager class. This will prevent other instances of ConfigurationFileManager from performing simultaneous file access, which could result thread safety problems.

1.4 Component Class Overview

ConfigurationFileManager

This is the central class of this component. It maintains an internal representation of the state of configuration represented in any number of files when those files were last accessed, and it keeps track of which configuration belongs to which file.

It relies on implementations of ConfigurationPersistence to actually read and write configuration to files. Two implementations of this interface are included in this component, which read and write files compatible with those defined in the ConfigurationFileManager component, but the user may define additional configuration files without difficulty.

This class is mutable and not thread safe.

FileType

This class contains public constants for each recognized file format. New file formats can be specified by extending the class.

It has two members: XML and PROPERTY. These represent the .xml and .property file formats used by the TopCoder Configuration Manager component.

Each file format has an associated fileExtension, which is set in the constructor. This file extension is used to determine the type of files specified by path in the ConfigurationFileManager configuration.

This class is immutable and thread safe.

ConfigurationPersistence

This interface defines the contract for classes that save and load ConfigurationObjects to and from files.

Two implementations of this interface are included with this component, one for xml files and one for property files. If a user needs support for some other format, a new implementation of this interface is needed and FileType should be extended.

The existing implementations of this interface are thread safe. It is not required that future implementations be thread safe.

PropertyFilePersistence

This implementation of ConfigurationPersistence may be used to save and read configuration from property files that are compatible with those defined in TopCoder Configuration Manager component. The functionality encapsulated in this class is very similar to functionality provided in the PropConfigProperties class from the Configuration Manager component.

This class is stateless and thread safe.

XMLFilePersistence

This implementation of ConfigurationPersistence may be used to save and read configuration from xml files of a format defined by the included Configuration_Manager_Schema.xsd, which is copied directly from the TopCoder Configuration Manager component. The functionality encapsulated in this class is very similar to functionality provided in the XMLConfigProperties class from the Configuration Manager component.

This class is stateless and thread safe.

1.5 Component Exception Definitions

1.5.1 Custom Exceptions

ConfigurationPersistenceException

This class is a base exception for the custom exceptions defined in this component. It is not thrown directly by any class.
This class derives from TopCoder BaseException, which is mutable and not thread safe. It is up to the application to handle exceptions in a thread safe manner.

InvalidConfigurationUpdateException
This exception is thrown by ConfigurationFileManager when an attempt is made to save a ConfigurationObject that includes descendent ConfigurationObjects that are not associated with any file in ConfigurationFileManager.
This class derives from ConfigurationPersistenceException, which is mutable and not thread safe. It is up to the application to handle exceptions in a thread safe manner.

ConfigurationUpdateConflictException
This exception is thrown by ConfigurationFileManager when an attempt is made to persist changes to configuration but some changes have been made to the underlying files since the last time they were loaded into or refreshed in ConfigurationFileManager.
This class derives from ConfigurationPersistenceException, which is mutable and not thread safe. It is up to the application to handle exceptions in a thread safe manner.

UnrecognizedFileTypeException
This exception is thrown by ConfigurationFileManager when an attempt is made to load configuration from a File with a FileType for which the ConfigurationFileManager has not been configured.
This class derives from ConfigurationPersistenceException, which is mutable and not thread safe. It is up to the application to handle exceptions in a thread safe manner.

UnrecognizedNamespaceException
This exception is thrown by ConfigurationFileManager when a namespace used to identify some part of the overall configuration doesn't correspond to any ConfigurationObject in the currentConfituration tree..
This class derives from ConfigurationPersistenceException, which is mutable and not thread safe. It is up to the application to handle exceptions in a thread safe manner.

NamespaceConflictException
This exception is thrown by ConfigurationFileManager when an attempt is made to load more than one file into a single namespace.
This class derives from ConfigurationPersistenceException, which is mutable and not thread safe. It is up to the application to handle exceptions in a thread safe manner.

ConfigurationParserException
This exception is thrown by ConfigurationPersistence when an attempt is made to load configuration from a configuration file that is invalid for the specific implementation being used. It passed to the user by ConfigurationFileManager.
This class derives from ConfigurationPersistenceException, which is mutable and not thread safe. It is up to the application to handle exceptions in a thread safe manner.

1.5.2 System exceptions

IlegalArgumentException
This is thrown to indicate a null or illegal argument.
IOException
This is thrown to indicate a problem reading or writing a file.

1.6 Thread Safety

This component is not thread safe. Due to immutability, all of the classes defined in this component are thread safe except for ConfigurationFileManager. ConfigurationFileManager has two mutable members, files and currentConfiguration, and is not thread safe. It could be made thread safe by using SynchronizedConfigurationObjects rather than DefaultConfigurationObjects for the two above named members. It can be used in a thread safe manner by synchronizing access to the ConfigurationFileManager.

One of the main goals of this design is to place as few limits on its use as possible. By not incorporating synchronization into the design, the application is free to use it in an unsynchronized manner or to take responsibility for synchronization. This is a more flexible approach than synchronizing under every circumstance, though it does make it harder to use in a multithreaded environment.

2. Environment Requirements

2.1 Environment

- Java 1.4

2.2 TopCoder Software Components

Configuration API 1.0
This component provides file based persistence for Configuration API.

Base Exception 1.0
Used as the base class for the custom exceptions in this component.

2.3 Third Party Components

None were used.

3. Installation and Configuration

3.1 Package Name

com.topcoder.configuration.persistence

3.2 Configuration Parameters

As described in the algorithms section.

4. Usage Notes

4.1 Required steps to test the component

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

4.2 Required steps to use the component

None.

4.3 Demo

A customer has the following properties files in the classpath:
`Component1.xml`

```
<?xml version="1.0"?>
<CMConfig>
```

```
            <Config name="com.topcoder.yyy">
                <property name="h">
                    <value>valueh</value>
                </property>
                <property name="i">
                    <property name="j">
                        <value>valuej</value>
                    </property>
                </property>
            </Config>
            <property name="a">
                <value>valuea</value>
            </property>
            <property name="b">
                <value>valueb</value>
                <property name="c">
                    <value>valuec</value>
                </property>
            </property>
            <Config name="com.topcoder.xxx">
                <property name="f">
                    <value>valuef</value>
                    <property name="g">
                        <value>valueg</value>
                    </property>
                </property>
                <property name="e">
                    <value>valuee1</value>
                    <value>valuee2</value>
                </property>
            </Config>
        </CMConfig>
```

Component2.properties

```
a=valuea
f=valuef1;valuef2
b=valueb
e=valuee1;valuee2
h.g=valueg
b.c=valuec
h.g.i=valuei
```

ConfigurationFileManager.properties

```
com.topcoder.abc=Component1.properties
```

As part of an application, a customer will load, modify, and save the configuration, and add a new configuration file to the classpath.

```
        ConfigurationFileManager manager = new
ConfigurationFileManager("test_files/demopreload.properties");
        // retrieve all stored configuration
        Map config = manager.getConfiguration();
        // we can retrieve the configuration object for com.topcoder.abc
        ConfigurationObject root = manager.getConfiguration("com.topcoder.abc");
        ConfigurationObject defaultObj = root.getChild("default");
        ConfigurationObject xxxObj = root.getChild("com.topcoder.xxx");
        ConfigurationObject yyyObj = root.getChild("com.topcoder.yyy");
        Object bValue = defaultObj.getPropertyValue("b");
        // bValue should be valueb
        System.out.println("com.topcoder.xxx.b = " + bValue);

        Object cValue = defaultObj.getChild("b").getPropertyValue("c");
        System.out.println("com.topcoder.abc.b.c=" + cValue);
        // get multi-values
        Object eValues[] = xxxObj.getPropertyValues("e");
        for (int i = 0; i < eValues.length; ++i) {
            System.out.println("com.topcoder.yyy.e = " + eValues[i]);
        }

        // A new file can easily be loaded into the current configuration
        String newNamespace = "com.topcoder.zzz";
        manager.loadFile(newNamespace, "test_files/demo.properties");
        ConfigurationObject zzzObj = manager.getConfiguration("com.topcoder.zzz");
        ConfigurationObject newObj = new DefaultConfigurationObject("newProperty");
```

```
zzzObj.addChild(newObj);
newObj.setPropertyValue("demoProperty", "123");
// We can persist these changes using the manager
manager.saveConfiguration(newNamespace, zzzObj);

// refresh the manager
manager.refresh();
```

5. Future Enhancements

- None.