[TopCoder]

Configuration Manager 2.2 Component Specification

1. Design

All changes performed when synchronizing documentation with the version 2.1.5 of the source code of this component are marked with **purple**.

All changes made in the version 2.2 are marked with **blue**.

All new items in the version 2.2 are marked with **red**.

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 Configuration Manager component centralizes the management of and access to these files. The benefits of Configuration Manager include:

1) A centralized code base for reading and loading configuration files
2) A system-wide caching strategy of configuration details to limit performance bottlenecks
3) A common interface for accessing and updating application properties

The existing functionality of Configuration Manager supports the following functionality:

- Preload a list of configuration files.
- Add additional configuration files once the configuration manager has started.
- Reload configuration files into memory when properties are modified.
- Ability to refresh the Configuration Manager if configuration files are modified.
- Ability to store the properties data in .properties and .xml files..

Configuration Manager v2.1.4 extended the functionality to allow:

- Use a pluggable source of configuration data. The pluggable sources are defined in .config files that contain the name of classes representing that source and their initial parameters.
- Delete the namespaces from in-memory set of namespaces existing within Configuration Manager.
- Delete the properties and their values both from in-memory set of properties and persistent storage.
- Control the process of loading the namespaces and limit the permissions to load and modify namespaces by classes owning the namespaces only.
- Use properties nested directly or indirectly in other properties. This allows for the representation of properties as a tree with a single root and for the manipulation of sub-trees in “JNDI Context” like manner.
- Maintain the integrity of the underlying files including comments.

The Configuration Manager resolves the concurrent updates to namespace properties in following manner:

1) The temporary copy of the namespace properties should be created with ConfigManager.createTemporaryProperties() method.

2) Any modifications to properties should be made with addProperty(), setProperty(), removeProperty(), removeValue() and other methods.

3) When modifications are complete a ConfigurationManager.commit() should be invoked to store the modifications in persistent store and make them visible. This method attempts to lock the namespace before saving the properties. If namespace is already locked by another user/process the exception is thrown.

Features new in Configuration Manager 2.1.4

Nested Properties

Configuration Manager 2.1.4 provides the facility to nest properties within other properties. The properties can be organized in a tree-like structure with a single “root” property owning all properties that are in that namespace. The “root” property is a fictitious property having no name that is created with a new instance of Namespace. This “root” property serves as an entry-point to the whole property tree.

In order to reference the nested properties, compound names are supported by this version of Configuration Manager. A compound name is a dot-separated String representing the full tree path to the target property. For example, a String “com.topcoder.util.config.ConfigManagerFile” is a compound name of a property that should be treated as : “property ConfigManagerFile is nested within property config that is nested within property util that is nested within property topcoder that is nested within property com”.

Different properties may have nested properties with same names.

File Integrity

Configuration Manager 2.1.4 maintains the order of underlying files (either .properties or .xml files). This includes the order of comments and properties. When loading the properties from configuration files any comments preceding the property declaration are collected in a List and are attached to the property. During the loading of properties, their values are collected in another List. This structure enables the original sequence of properties to be maintained when the properties are saved back to the persistent store.

Pluggable Sources of configuration properties

Configuration Manager 2.1.4 facilitates the use pluggable sources of configuration data. To use such source the PluggableConfigSource interface must be implemented. The initial parameters needed to initialize the source should be placed in some file with a filename extension equal to “.config”. Also, a string like

“classname=<some class implementing PluggableConfigSource>”.

Implementations of PluggableConfigSource should adhere to the following:

1) They should provide a public non-argument constructor.
2) They should accept the initial parameters from java.util.Properties provided to configure() method.

Swappable implementation of Configuration Manager

Configuration Manager allows custom implementations of the Configuration Manager logic. To do so, the name of the concrete class that is derived from the ConfigManager class must be specified as the "implementor" property within the namespace owned by Configuration Manager. If such a class is not specified, is not accessible or any other error occurs while creating an instance of this class, then an instance of DefaultConfigManager is used. Note that this feature for some reason was missing in the source code of the versions 2.1.4 and 2.1.5 (though it was even mentioned in the javadoc). So this feature is actually implemented in the version 2.2 only.

Escaping in ".properties" configuration files

Configuration Manager 2.1.4 supports escaping these characters: \#, \!, \=, \ (slash-space), \\, \:, \n, \r, \t, \uXXXX. For example, supposed there is the following item in ".properties" file:

prop\ \\\:\n\r\t\u0020=value\#\!\=

when it is parsed, key is this java string "prop \\:\n\r\t ", value is this java string "value#!=", note that \u0020 is parsed to a space.

In the version 2.2 the following changes were performed:

- Changed development language to Java 1.5.
- ConfigManager#getInstance() method was made synchronized to properly implement thread safe singleton pattern.
- save() and load() methods of ConfigProperties subclasses were made synchronized to avoid thread safety issues when Configuration Manager is used from multiple threads and configuration properties need to be refreshed.
- Implemented "Swappable implementation of Configuration Manager" feature (see above) that was mentioned in the documentation and javadoc, but has been never provided in the source code.
- Now it's possible to specify in the configuration whether a namespace is refreshable or not. If namespace is not refreshable, it's configuration properties are not reloaded from persistence when refresh() or refreshAll() method of DefaultConfigManager is called. It's possible to specify whether configuration parameters are refreshable or not by default in the configuration of ConfigManager.

FOR DEVELOPERS: Please note that the development language was changed to Java 1.5, so it's required to specify generic parameters for all generic types in the source code. Please consult "Implementation notes" section of method docs in TCUML if it's not clear what type should be used as a generic parameter. (Note that methods in which generic types were specified in the source code, but not signature, are not marked with blue in TCUML). Additionally developers can remove statements that check types of elements obtained from generic collections. Also there is no need to cast elements obtained from the generic collections anymore.

1.1 Design Patterns

Singleton pattern – ConfigManager implements this pattern.

Strategy pattern – PluggableConfigProperties uses pluggable PluggableConfigSource instances.

1.2 Industry Standards

XML, JAXP

1.3 Required Algorithms

All algorithms used in this component are quite simple. Please see "Implementation notes" sections of method docs provided in TCUML for details.

1.4 Component Class Overview

ConfigManager [abstract]

The ConfigManager class centralizes the management of and access to applications' configuration details. It provides a common interface for accessing and updating applications' properties.

The ConfigManager provides a possibility to maintain a list of applications' properties through standard .properties files, an .xml files based on Configuration Manager DTD, an .xml files containing multiple namespaces. Since version 2.1 a pluggable sources of configuration properties are also available. Those sources are described in .config files that represent a standard properties files. The .config files should contain a required classname property that contains the name of class implementing the PluggableConfigSource interface, and any number of key-value pairs that should be used to configure this PluggableConfigSource instance.

The public methods provided by ConfigManager allow to:

- query the value of properties of namespaces loaded into Configuration Manager

- add new namespaces to Configuration Manager

- modify the properties of existing namespaces

- remove namespaces from memory

- remove properties and their values from namespaces

- nest the properties within other properties as deep as needed

The ConfigManager has its own configuration file named com/topcoder/util/config/ConfigManager.properties that should be placed in directory which CLASSPATH system environment variable points to. This file contains the definitions of namespaces that should be loaded into memory with every start of Configuration Manager.

The methods of ConfigManager modifying the namespaces, such as: addToProperty(), setProperty(), removeProperty(), removeValue() are executed only after a temporary copy of namespace has been created. Any changes made by those methods are not visible until the commit() method is invoked.

Changes in 2.2:

- Made a full class name of ConfigManager subclass to be used as a singleton configurable (see "implementor" configuration parameter).

- Added a property indicating whether namespace configuration parameters are refreshable by default (this property is read from the class configuration).

ConfigManagerInterface [interface]

This is an interface that all classes which deal with the ConfigManager (including ConfigManager itself) should implement.

ConfigProperties [abstract]

This abstract class is the superclass of all classes representing the different sources of configuration properties. While this class maintains the properties data its subclasses are responsible for interacting with underlying persistent storage to save and load properties data.

Changes in 2.2:

- Added refreshable property.

- Made this class implement Cloneable (method clone() has been already present).

DefaultConfigManager

This is a default implementation of ConfigManager's logic that is renamed ConfigManager class from previous version of TC Configuration Manager.

Changes in 2.2:

- Added support for not refreshable configuration properties to refreshAll() and refresh() methods.

Helper

This is helper utility class used by this component. It defines methods that are called by multiple classes from the component.

Namespace

A namespace existing within Configuration Manager. Is described with name uniquely identifying it among other namespaces, format of source containing the properties of namespace, URL pointing to source containing the properties of namespace, level of exceptions and ConfigProperties associated with it.

Before attempting to make any updates to namespace a namespace should be locked first with lock() method. After namespace has been locked and all updates have been made a namespace should be unlocked with unlock() method. canLock() can be used to check whether this namespace can be locked by given user or not.

PluggableConfigProperties

An extension of ConfigProperties providing the ability to use pluggable sources of configuration properties.

Changes in 2.2:

- Made save() and load() methods synchronized.

- Added support for "IsRefreshable" configuration parameter.

PluggableConfigSource [interface]

A pluggable source of configuration data. An implementations of this interface should have a public non-argument constructor and should support their configuration with Properties.

PropConfigProperties

An extension of ConfigProperties to be used to maintain the properties list using .properties files.

Changes in 2.2:

- Made save() and load() methods synchronized.

- Added support for "IsRefreshable" configuration parameter.

Property

An element of configuration data representing some property. A property may have single or multiple values and zero or more nested subproperties. A property is described with name uniquely identifying it among other properties. While Property has public get-interface it's set-interface is hidden from user. The content of existing property can be manipulated only with use of appropriate ConfigManager's methods. However, this class provides a set of public constructors sufficient to create a property with given name and value or array of values.

The Property object serves as a "root"-context for all properties nested directly or indirectly within it. The methods manipulating with property names accept a compound name pointing to any property nested within this Property. A compound name is a dot-separated String each part of which represents a name of a single level property. For example : a property name "countries.USA.currency" points to property named "currency" that is nested within property named "USA" that is nested within property named "countries".

XMLConfigProperties

An extension of ConfigProperties to be used to maintain the properties list using .xml files.

Changes in 2.2:

- Made save() and load() methods synchronized.

- Added support for "IsRefreshable" configuration parameter.

1.5 Component Exception Definitions

ConfigLockedException
Exception to be thrown when there is an attempt to lock a namespace but it is already locked by another user.

ConfigManagerException
Generic Exception thrown by the Config Manager. Thrown when errors specific to the Config Manager occurs.

ConfigParserException
Thrown when any exception preventing normal extraction of configuration data occurs. Such exceptions are thrown by subclasses of ConfigProperties during their interaction with underlying source of configuration data.

DuplicatePropertyException
Thrown when there is an attempt to add a nested property to Property with name that already exists within the Property.

NamespaceAlreadyExistsException
Thrown when there is an attempt to add a Namespace which already exists in ConfigManager.

UnknownConfigFormatException
Thrown when specified format of configuration properties is not in list of valid values, those are: ConfigManager.CONFIG_XML_FORMAT, ConfigManager.CONFIG_MULTIPLE_XML_FORMAT, ConfigManager.CONFIG_PROPERTIES_FORMAT, ConfigManager.CONFIG_PLUGGABLE_FORMAT.

UnknownNamespaceException
Thrown from any of ConfigManager's methods (except existsNamespace() that is invoked with namespace that does not exist within ConfigManager).

1.6 Thread Safety

The version 2.1.5 of the component was almost fully not thread safe. DefaultConfigManager just used synchronized namespaces and tempProperties collections that is definitely not enough to make the class thread safe. DefaultConfigManager uses not thread safe Namespace and ConfigProperties instances that makes this class not thread safe.

In the version 2.2 ConfigManager#getInstance() method was made synchronized to implement thread safe singleton pattern. Also save() and load() methods of ConfigProperties subclasses are made synchronized, so now it's safe to call refresh() and refreshAll() methods of DefaultConfigManager from multiple threads at a time. Assuming that other methods (except refresh() and refreshAll()) are not used, it's safe to additionally call property value retrieval methods from multiple threads.

Other functionality of DefaultConfigManager that can modify its internal state was not change, so it stays to be not thread safe.

2. Environment Requirements

2.1 Environment

Development language: Java1.5
Compile target: Java1.5

2.2 TopCoder Software Components

None

2.3 Third Party Components

None

3. Installation and Configuration

3.1 Package Name

com.topcoder.util.config

3.2 Configuration Parameters

com/topcoder/util/config/ConfigManager.properties

This file should be placed into directory accessible from CLASSPATH. It should contain the definitions of namespaces that should be loaded each time the Configuration Manager starts. The format of these definitions is as follows:

<namespace> = <file>, for example :

com.topcoder.util.config.ConfigManager = com/topcoder/util/config/CM.properties

Additionally starting from the version 2.2 this file can contain the following parameters:

- **implementor** – the full class name of ConfigManager to be used as a singleton instance. Default is “com.topcoder.util.config.DefaultConfigManager”. If custom configuration manager cannot be created for some reason, DefaultConfigManager is used.
- **refreshableByDefault** – the value indicating whether configuration properties should support refreshing by default (“true”, ignoring case, means “yes”; all other values mean “no”). Default is “yes”.

E.g.:

implementor = com.my.CustomConfigManager
refreshableByDefault = false

It’s possible to override “refreshableByDefault” behavior for each ConfigProperties instance (i.e. for each namespace) separately. To do so “IsRefreshable” property should be specified in the configuration (“true”, ignoring case, means “yes”; all other values mean “no”).

If the configuration for specific is not refreshable, this means that in case if DefaultConfigManager#refresh() or DefaultConfigManager#refreshAll() is called by the user, the configuration properties for this namespace won’t be reloaded from the persistence.

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

Please see the demo.

4.3 Demo

4.3.1 API usage

```
// First an instance of ConfigManager should be obtained. This will also
 // load the predefined set of namespaces and their properties into  memory
 ConfigManager manager = ConfigManager.getInstance();

 // Check if configuration is refreshable by default
 boolean configRefreshableByDefault = manager.isConfigRefreshableByDefault();

 // Set configuration refreshable by default to "false"
 manager.setConfigRefreshableByDefault(false);

 // any additional namespace may be loaded then
 manager.add("com.topcoder.currency", "currency.xml", ConfigManager.CONFIG_XML_FORMAT);

 // once the namespace properties are loaded they can be queried
 String value = manager.getString("com.topcoder.currency", "countries.USA.currency");

 value = manager.getString("com.sample.first", "property1");
 // value must be equal to "Configuration Manager"

 // multiple values of same property are also supported
 String[] values = manager.getStringArray("com.topcoder.currency", "countries.USA.name");

 // a new properties may be defined and stored for further use
 manager.createTemporaryProperties("com.topcoder.currency");
 manager.setProperty("com.topcoder.currency", "countries.Germany", "value");
 manager.addToProperty("com.topcoder.currency", "countries.USA.currency.name", "US dollar");
 manager.commit("com.topcoder.currency", "user");

 // removing of properties
 manager.createTemporaryProperties("com.topcoder.currency");
 manager.removeProperty("com.topcoder.currency", "countries.Utopia");
 manager.commit("com.topcoder.currency", "user");

 // removing of values of properties
 manager.createTemporaryProperties("com.topcoder.currency");
 manager.removeValue("com.topcoder.currency", "countries.USA.currency.name", "US dollar");
 manager.commit("com.topcoder.currency", "user");

 // refresh one namespace (if it's refreshable)
 manager.refresh("com.topcoder.currency");

 // removing namespaces from in-memory set of loaded namespaces
 manager.removeNamespace("com.topcoder.currency");

 // checking the permission of current thread to load and modify namespaces
 try {
    manager.add("namespace.owned.by.some.class", "somefile.xml", ConfigManager.CONFIG_XML_FORMAT);
    manager.createTemporaryProperties("namespace.owned.by.some.class");
    manager.setProperty("namespace.owned.by.some.class", "new.property", "value");
    manager.commit("namespace.owned.by.some.class", "user");
 } catch (ConfigLockedException e) {
    System.err.println("Hey! Ask a namespace owner to load or " + "modify the namespace");
 }

 // addition of pluggable sources of configuration properties
 manager.add("pluggable.namespace", "somefile.config", ConfigManager.CONFIG_PLUGGABLE_FORMAT);

 // Demo for escaping in ".properties" files, supposed the
 manager.add("EscapeEnhancement", "test_files/TestEscape.properties",
```

```
ConfigManager.CONFIG_PROPERTIES_FORMAT);
        // string s is "hello"
        String s = manager.getString("EscapeEnhancement", " \n\rProp3 ==!\t\\:!# ");

        // refresh all refreshable namespaces
        manager.refreshAll();
```

4.3.2 Sample configuration

com/topcoder/util/config/ConfigManager.properties

```
# configuration manager properties
implementor = com.topcoder.util.config.DefaultConfigManager
refreshableByDefault = false
# list of pre-load namespaces
com.sample.first=config1.properties
com.sample.second=config2.xml
```

config1.properties

```
ListDelimiter=#
IsRefreshable=false
property1 = Configuration Manager
property2 = foo#bar#baz
property3 = 4#5#6
# Comment
prop4 = "ASDFASDFAF"
```

config2.xml

```
<?xml version="1.0"?>
<CMConfig>
    <IsRefreshable>true</IsRefreshable>
    <Property name="first">
        <Value></Value>
    </Property>
    <Property name="Property1">
        <Value>test2</Value>
    </Property>
    <Property name="Property2">
        <Value>4</Value>
    </Property>
    <Property name="Property5">
        <Value>MORE</Value>
    </Property>
</CMConfig>
```

For the sample configuration provided above in case if refreshAll() method is called, only configuration from “config2.xml” file will be reloaded by the configuration manager.

4.3.3 Usage from multiple threads

Starting from the version 2.2 this component is partially thread safe.

```
/**
 *
 * <p>
 * An implementation of Runnable.
 * </p>
 *
 * @author TCSDEVELOPER
 * @version 2.2
 * @since 2.2
 */
Runnable runnable = new Runnable() {
    /**
     * <p>
     * The <code>run</code> method to be called in that separately executing thread.
     * </p>
     */
    public void run() {
```

```
        for (int i = 0; i < ITERATIONS; i++) {
            // Refresh
            refresh();
        }

        sleep(1000);

        System.out.println("runnable finished");
    }
};
Thread thread1 = new Thread(runnable);
Thread thread2 = new Thread(runnable);
thread1.start();
thread2.start();

sleep(5000);
```

The output would be:

```
 finished refreshing
  finished refreshing
  …
  … (2* ITERATIONS times)
  runnable finished
  runnable finished
```

5. Future Enhancements

None