.NET Configuration Manager 2.0 Component Specification

1. Design

The design is simple, yet flexible, and is guided by the logical structure of the properties files. You can find the ConfigFile class that corresponds to a configuration file. As expected, a ConfigFile contains several Namespace class instances that correspond to the namespaces in the XML configuration files (namespaces == sections for INI files). The Namespace contains several Property class instances that correspond to the actual properties.

All these classes provide easy navigation between them achieving a sort of introspection ability. C# properties allow easy access to names and values.

The main class however is the ConfigManager. It has the responsibility of knowing which namespaces are loaded and load new configuration files. It also provides shortcut methods to get a namespace, a property and a value or set a value in one call.

The ability to handle multiple configuration file formats is implemented using a strategy like pattern. The FileHandler interface abstracts a loader/saver for a configuration file format. The ConfigFile class will choose from the available implementations the right one for a given file based on extension (can be easily changed to a more complex check in the Supports method but for now extensions will do).

The preloading of a configurable list of files is covered in the algorithms and configuration sections.

A requirement was to allow the user to specify the errors that should be ignored. This is covered in detail in the algorithms section.

To fulfill the new requirements, the system wide singleton of ConfigManager is substituted by normal singleton, and new functionalities are added to the ConfigManager to get values of specified type, get values following user defined parsing preference, and create instances from the type/assembly property in the datasource. And ConfigManager also supports a new kind of file extension, which is used by the PluggableHandler to create IDataHandler to load/save namespaces to pluggable datasource.

New requirements:

1. Remove the System-Wide-Singleton

   The System Wide Singleton is replaced with normal singleton pattern.

2. Extend the concept of ‘Config Files’ to include other data sources

   A new kind of configuration files (its file extension is ‘mxml’) is supported by the ConfigManager, which can be processed by the PluggableHandler to load/save namespaces from a pluggable datasource. The mxml defines necessary information to instantiate the IDataHandler instance that is responsible the namespace load and save.

   The preloading of preload configuration file from the app.config is already supported by the version 1.1.

3. Improve the component interface

   User of ConfigManager can call GetValue/GetValues methods with an IParsingPreference instance so that the user will process the retrieved value from the

datasource with defined parsing preference first before returning it. If the value violates any of the preference rules, ParsingPreferenceException will be thrown.

User of ConfigManager can call GetValue/GetValues methods with IStringConverter to convert the retrieved string value into specified type. DefaultStringConverter is used by default if no IStringConverter instance is specified, which can convert string to the other primitive types and Enum values correctly.

User of ConfigManager can also call CreateInstance methods to create the instance from the type and/or assembly property.

4. Retain API compatible

   None of the old API is changed, so it is definitely compatible with the old version.

NOTE: the red contents are newly added.

1.1 Design Patterns

- Singleton (the ConfigManager class)
- Strategy (the FileHandler class, IParsingPreference and IStringConverter interfaces.)

1.2 Industry Standards

XML.

1.3 Required Algorithms

1.3.1 The XML format for the configuration files

The XML file used to store signatures has the following DTD:

```
<!ELEMENT ConfigManager (namespace+)>
<!ELEMENT namespace (property+)>
<!ELEMENT property (value+)>
<!ELEMENT value (#PCDATA)>
<!ATTLIST namespace
      name NMTOKEN #REQUIRED
>
<!ATTLIST property
      name NMTOKEN #REQUIRED
>
```

The format is very simple. The root is the ConfigManager element. The root can contain one or more namespaces as the requirements say, each namespace can contain one or more properties and each property can contain one or more values. The namespaces and the properties are identified by names.

Here is a sample:

```
<?xml version="1.0" encoding="UTF-8"?>
<!doctype ConfigManager SYSTEM "cm.dtd">

<ConfigManager>
      <namespace name="TopCoder.Util.ConfigurationManager">
            <property name="preload">
                  <value>..\..\test_files\test1.xml</value>
                  <value>..\..\test_files\test2.xml</value>
            </property>
```

```
        </namespace>
</ConfigManager>
```

1.3.2 Parsing the XML configuration file:

To parse the XML configuration file the XmlValidatingReader should be used. The process is rather simple. Keep getting nodes using the pull model. When a namespace node is encountered a Namespace object is instantiated. When a property node is encountered a Property object is instantiated, after all the children Value nodes are processed.

The writing of XML files should be done using the XmlWriter class. All the information is available from the Namespace and Property classes using the exposed properties. The process is straight forward also considering the simplicity of the DTD.

1.3.3 *Error tolerance:*

The ErrorCallback class allows the user to choose the handling of error on a case by case basis. When an error occurs (namespace collision, property collision or critical error) the corresponding method in the callback instance provided by the user is called, and an action is taken as indicated by the user. Note that collision means having the same name. The ErrorAction enumeration contains values for the actions to be taken (replace old namespace/property, keep old namespace/property, merge namespaces/properties, throw an appropriate exception.

1.3.4 The preload list:

The ConfigManager private constructor has the task of loading the preloadFile and extract from there all the names of the configuration files that need to be preloaded. Then each file will be loaded. See the Configuration Parameters section for more details.

1.3.5 Synchronization issues under concurrent environment:

The following accesses need to be synchronized, considering the concurrency requirements, by enclosing the code parts which access them in lock blocks with the owner class as lock objects:

- The modification of the Property values field
- The modification of the Namespace properties and propertiesLookup fields
- The modification of the ConfigManager namespaceLookup field
- The save, save to, remove and reload operations need to lock the ConfigManager instance to prevent them executing concurently

1.3.6 Loading a configuration file in a new namespace

The ConfigManager class provides methods to load a configuration file into a new newspace. This means that the configuration file will be read and all properties will go into the new namespace, ignoring any namespace information contained in the file.
If a file has 3 namespaces, ns1, ns2 and ns3, and the new namespace is ns, all the properties from ns1, ns2 and ns3 will be placed in ns. The namespaces ns1, ns2 and ns3 won't even be created.

1.3.7 Reading the INI configuration file:

The INI configuration file is a text file structure on lines. The section lines have the form:

[section_name]

When such a line is encountered a new Namespace object is instantiated.

After each section declaration lines in the form:

property_name = property_value1 [;property_value2;….]

such lines should be split into its components and a Property object should be instantiated.

The writing of INI files is even simpler. Simply iterate the namespaces and output the section line. An inner loop should iterate the properties of each namespace and output the property lines.

1.3.8 Reading the mxml for the PluggableHandler

Here is the format of mxml file (which is actually an xml file, prepend a 'm' character in order to tell it apart from the normal xml file).

```
<?xml version="1.0" ?>
<PluggableHandler>
	<Class>TopCoder.Util.ConfigManager.DatabaseHandler</Class>
	<!-- assembly is optional -->
	<Assembly>xxx</Assembly>
	<Properties>
		<Property name="connectionUrl">
			<Value>xx</Value>
		</Property>
		<Property name=”namespaceTable”>
			<Value>namespaces_tbl</Value>
		</Property>
		<Property name=”propertyTable”>
			<Value>properties_tbl</Value>
		</Property>
		<Property name=”valueTable”>
			<Value>values_tbl</Value>
		</Property>
	</Properties>
</PluggableHandler>
```

When loading the mxml with ConfigManager.LoadFile method, PluggableHandler will be used to parse the file to create the IDataHandler instance from the class and assembly properties, and the properties node will populate a map of key-value pairs to pass into the IDataHandler subclass' constructor. Implementation of IDataHandler should always provide a constructor taking an IDictionary.

Refer to the demo section for a sample implementation of the DatabaseHandler, though the implementation is not complete, it should be enough to guide the developer to understand how to implement the IDataHandler interface.

1.4 Component Class Overview

Class ConfigManager:

This is the main class, which provides the interaction with the user (a singleton). The class exposes methods for loading new configuration files and getting information about loaded namespaces, properties and their values. It also allows changing the values of the properties, removing, reloading and saving configuration files.

Class ConfigFile:

This class represents a configuration file and stores internally all the information needed to write back the configuration file if needed. The file can be written in the same format of the original or in any other supported format.

Class Namespace:
A namespace groups logically a set of properties. As expected, the namespace contains a set of Property objects and properties and methods to access them.

Class Property:
This class corresponds to a property. A property has a name and more string values. The class exposes properties to access the name and the values. Only the values are modifiable.

Interface FileHandler:
This interface abstracts a configuration file handler. A configuration file handler has the task of identifying whether it can process the file, loading namespaces with their properties from a file and saving a set of namespaces to a file..

Class XmlFileHandler:
A configuration file handler for XML files as they are defined in the component specification.

Class IniFileHandler:
A configuration file handler for INI files.

Enumeration ErrorAction:
This enumeration provides values through which the user can communicate the desired behavior when a namespace or property collision occurs.

Interface ErrorCallback:
Instances of implementations of this interface will be passed by the user to specify custom handling of the errors that can occur while loading a config file (namespace clash, property clash, critical errors). When an error occurs the corresponding method in the callback instance is called to get the desired action to be taken from the user.

Class DefaultErrorCallback:
Default implementation of the ErrorCallback interface that chooses exception throwing for all errors.

Class PluggableHandler
It is responsible of parsing the mxml file to create the IDataHandler instance, which is responsible for loading/saving the namespace from the datasource.

Interface IDataHandler
IDataHandler interface is responsible for loading/saving the namespaces from the datasource. The PluggableHandler will create it dynamically.

Interface IParsingPreference
IParsingPreference interface will be used by ConfigManager to define user defined preference rule, and the retrieved string value from configuration manager should obey the rule.

Class ParsingPreference
ParsingPerference implements IParsingPreference interface and provide basic rules such as null/empty string is allowed or not, white space will be trimmed or not.

Interface IStringConverter

IStringConverter interface will be used by ConfigManager to convert string value into specified type.

Class DefaultStringConverter

DefaultStringConverter implements IStringConverter interface, and is used by ConfigManager by default to convert string into primitive types or Enum value.

1.5 Component Exception Definitions

Exception NamespaceClashException:

Exception thrown when a namespace name conflict occurs (a namespace is loaded and a namespace with the same name already exists).

- ConfigManager constructor
- ConfigManager LoadFile methods
- ConfigFile constructor
- ConfigFile Reload method
- FileHandler and its subclasses LoadFromFile methods

Exception PropertyClashException:

Exception thrown when a property name conflict occurs (a property is loaded and a property with the same name already exists in the owner namespace).

- ConfigManager constructor
- ConfigManager LoadFile methods
- ConfigFile constructor
- ConfigFile Reload method
- FileHandler and its subclasses LoadFromFile methods
- Namespace AddProperty method

Exception InvalidConfigFileException:

Exception thrown when an error occurs while loading a configuration file (signaling the file is invalid). If the error is caused by another exception (XML parsing exceptions for example) then the other exception is wrapped as inner exception. I/O exceptions are not included here. They should be propagated.

- ConfigManager constructor
- ConfigManager LoadFile methods
- ConfigFile constructor
- ConfigFile Reload method
- FileHandler and its subclasses LoadFromFile methods

Exception UnknownFormatException:

Exception thrown when the configuration file format is unknown (no handler supports it).

- ConfigManager constructor
- ConfigManager LoadFile methods
- ConfigFile constructor

- ConfigFile Reload and SaveTo method
- FileHandler and its subclasses LoadFromFile methods

Exception IOException:

This exception is generated by all the classes when an I/O error occurs while working with files.

- ConfigManager constructor
- ConfigManager LoadFile methods
- ConfigFile constructor
- ConfigFile Reload, Save, SaveTo method
- FileHandler and its subclasses Supports, LoadFromFile and SaveFile methods

Exception MergeException:

Thrown when the configuration file cannot be saved or unloaded because some of its namespaces or properties were merged, so it is unsafe to save or unload it as the saved config file will be mixed with data from other config files or the removal might corrupt other config files loaded in memory. It is thrown by ConfigManager.ReloadFile/RemoveFile/SaveFile/SaveFileTo and ConfigFile.Save/SaveTo.

ArguemntNullException:

This exception is thrown by all the public methods when any of the arguments are null. The arguments that are arrays need to be checked for null at element level also. The zargo doc and xml doc details each case.

ArgumentException

This exception will be thrown if the string value is empty, refer to the documentation tab for more details.

FormatException

This exception is thrown if the string value cannot be converted to the specified type correctly.

InstanceCreationException [Custom]

This exception is thrown if CreateInstance methods of ConfigManager fail to create the instance from the given properties.

DataHandlerException [Custom]

This exception is thrown if the IDataHandler fails to load/save namespaces from the related datasource.

ParsingPreferenceException [Custom]

This exception is thrown if the string passed to IParsingPreference does not follow the user defined preference rules.

1.6 Thread Safety

This component should be thread-safe definitely, and it is always possible to be used in multi-threaded environment. When loading file into the ConfigManager, the namespaces retrieved from the datasources will be copied and stored in a hash table, which should be

synchronized. And either load or save namespaces to the datasource will finally use the specific FileHandler instance to really persist the data. So it should be guaranteed that the FileHandler implementation should be thread-safe when operating on the persistence. So do the IDataHandler implementations.

2. Environment Requirements

2.1 TopCoder Software Components:

- None

2.2 Third Party Components:

- None

3. Installation and Configuration

3.1 Namespace

TopCoder.Util.ConfigurationManager

3.2 Configuration Parameters

The list of configuration files to be loaded by default at instantiation can be configured using the conf/preload.xml configuration file (Configuration Manager style). Under the namespace TopCoder.Util.ConfigurationManager the property preload enlists the files to be preloaded. Each value designated the name of a file.

The current configuration file has the following demo contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<!doctype ConfigManager SYSTEM "cm.dtd">

<ConfigManager>
        <namespace name="TopCoder.Util.ConfigurationManager">
                <property name="preload">
                        <value>..\..\test_files\test1.xml</value>
                        <value>..\..\test_files\test2.xml</value>
                </property>
    </namespace>
</ConfigManager>
```

3.3 Dependencies Configuration

None.

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Execute ‘nant test’ within the directory that the distribution was extracted to.

4.2 Required steps to use the component

The usage is very simple, just get an instance of the ConfigManager by calling ConfigManager.getInstance(). Simple usage required two additional method calls. One to load a file (LoadFile) and the other to get the value of a property (GetValue).

More advanced usage allows the modification of properties (SetValue) and the saving of the modified properties back to disk (using the class ConfigFile returned to the caller of LoadFile, the methods Save or SaveTo).

Another nice feature is the introspection of the properties. The user can browse the available namespaces and their properties freely.

The following code demonstrates how to use this component in another component to load configuration values.

If the component need to load the following properties from the 'TopCoder.Example' namespace:

property name	property value data type
username	string type
age	int type
salary	double type

It should load those values as follows:

```
string ns = "TopCoder.Example";

ConfigManager cm = ConfigManager.GetInstance();

// refresh the namespace in order to reload the newest data

cm.Refresh();

// get all the properties

string username = cm.GetValue(ns, "username");

int age = cm.GetIntValue(ns, "age");

double salary = cm.GetDoubleValue(ns, "salary");
```

If you load the properties from the xml file, it should like the one below: (assume its name is example.xml)

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

<namespace name="TopCoder.Example">
<property name="username">
   <value>topcoder</value>
</property>
<property name="age"
   <value>12</value>
</property>
<property name="salary">
   <value>13.3</value>
</property>
</namespace>
</ConfigManager>
```

Then you should add a reference to this file in the preload.xml like follows:

```
<?xml version="1.0"?>
<ConfigManager>
   <namespace name="TopCoder.Util.ConfigurationManager">
     <property name="preload">
       <value>example.xml</value>
     </property>
```

```
   </namespace>
</ConfigManager>
```

It is rather similar when loading the properties from the other type of data sources.

4.3 Demo

Here are some code samples:

1. Load a configuration file and get the value of a property from a given namespace:

```
ConfigManager cm = ConfigManager.GetInstance();

ConfigFile cf = cm.LoadFile("your_xml_file.xml");
     or
ConfigFile cf = cm.LoadFile("your_ini_file.ini");

Console.WriteLine(cm.GetValue("namespace1", "property1"));
String[] values = cm.GetValue("namespace1",
                              "multiple_value_property2")
Console.WriteLine(values.Length);

cm.RemoveFile("your_ini_file.ini"); // cleanup if necessary
```

2. Load a configuration file change the value of a property and save it back to disk:

```
ConfigManager cm = ConfigManager.GetInstance();

ConfigFile cf = cm.LoadFile("your_xml_file.xml");
     or
ConfigFile cf = cm.LoadFile("your_ini_file.ini");

cm.SetValue("namespace1", "property1"));
cm.SetValue("namespace1", "property2", new String[] {"a", "b"});

cm.SaveFile("your_xml_file.xml"); // save over original
cm.SaveFileTo("your_xml_file.xml",
              "new_xml_file.xml"); // save to new xml
cm.SaveFileTo("your_xml_file.xml",
              "new_ini_file.ini"); // save to new ini
```

3. Introspect loaded namespaces and values:

```
ConfigManager cm = ConfigManager.GetInstance();
ConfigFile cf = cm.LoadFile("your_xml_file.xml");

foreach (Namespace ns in cm.Namespaces)
```

```
{
    Console.WriteLine(ns.Name);
    foreach (Property p in ns.Properties)
    {
        Console.WriteLine(p.Name + " = " + p.Value);
    }
}
```

```
4. Get Value of specific type
// Using the DefaultStringConverter to convert the string value.

// if the property 'prop1' of namespace 'ns1' contains
// a value '12', call with typeof(int) will convert the
// the string value to an int.
int value = (int) cm.GetValue('ns1', 'prop1', typeof(int));

// if the property 'prop2' of namespace 'ns1' contains
// an list of value like '11', '12', '13', call with typeof (int)
// will return an int array.
int[] values = (int[]) cm.GetValues('ns1', 'prop2', typeof(int));

// if the property 'prop3' of namespace 'ns1' contains
// a value 'Red', which is one value of the Enum class:
Enum Color { Red, Blue, Green };
// call with typeof(Enum) will convert it to a Color instance
Color red = (Color) cm.GetValue('ns1', 'prop3', typeof(Color));

// And client may add his own IStringConverter implementation to
// convert the string to fulfill his needs.
// The following implementation will convert the null value to
// the valid value to different types.
class MyStringConverter : IStringConverter {
      public object Convert(string value, Type type) {
            if (value == null) {
                  if (type == typeof(int)) {
                        return 0;
                  } else if (type == typeof(double)) {
                        return 0.0;
                  } else {
                        throw NotSupportedException();
                  }
            }
     return DefaultStringConverter.Instance.Convert(value, type);
      }
}
// and it can be used as follows:
int value = (int) cm.GetValue('ns1', 'non-exist', typeof(int));
// although the property does not exist (null string will be
// retrieved), the null will be converted to a 0 finally.

5. Get Value with parsing preference
// User can have different preferences
// Null value will cause exception by this preference
ParsingPreference pref1 = new ParsingPreference();
pref1.AllowNull = false;
```

```
// Empty value will cause exception by this preference
ParsingPreference pref2 = new ParsingPreference();
pref2.AllowEmpty = false;

// White spaces will be trimmed by this preference
ParsingPreference pref3 = new ParsingPreference();
pref3.TrimSpace = true;


// call with pref1 if the property does not exist (null value)
// will cause exception.
try {
      cm.GetValue('ns1', 'non-exist', pref1);
      fail();
} catch (ParsingPreferenceException ppe) {
      // good
}

// call with pref2 if the property contains empty value will
// cause exception.
try {
      cm.GetValue('ns1', 'empty-prop', pref2);
      fail();
} catch (ParsingPreferenceException ppe) {
      // good
}

// call with the pref3, white spaces of the returned value will
// be trimmed.
String value = cm.GetValue('ns1', 'white-space-prop', pref3);
Assertion.AssertTrue(value.equals(value.trim()));

// Client can implement the IParsingPreference interface to
// fulfill his own needs.
// The following implementation restrict the value should be
// present in an string array.
class MyParsingPreference implements IParsingPreference {
      private String[] validValues = null;
      public MyParsingPreference(String[] validValues) {
            this.validValues = validValues;
      }
      public string Parse(string value) {
            if (Array.IndexOf(value, validValues) >= 0) {
                  return value;
            } else {
                  throw ParsingPreferenceException("xx");
            }
      }
}
MyParsingPreference mpp = new MyParsingPreference(
       new String[] { "Red", "Blue", "Green" } );
// the Color property should only contain value from
// the string array above.
string value = cm.GetValue('ns1', 'Color', mpp);
```

```
6. Create instances with the type/assembly property
// Here is an interface and an implementation of the interface
// which provides a empty constructor and a constructor taking
// two arguments.
interface ITest {
      void TestIt();
}

class Test : ITest {
      public Test() {}
      public Test(string a, int b) {}
      public void TestIt() {
            // nothing
      }
}

// Create instance with typename property and no arguments ctor
ITest test =
     (ITest) cm.CreateInstance('ns1', 'TestClass', typeof(ITest));

// Create instance with typename property and ctor with args
ITest test =
     (ITest) cm.CreateInstance('ns1', 'TestClass',
      new object[] { 'a', 12 }, typeof(ITest));

// Create instance with typename property, assembly name property
// and no arguments ctor
ITest test =
      (ITest) cm.CreateInstance('ns1', 'TestClass',
            'TestAssembly', typeof(ITest));

// Create instance with typename property, assembly name property
// and ctor with arguments.
ITest test =
      (ITest) cm.CreateInstance('ns1', 'TestClass',
        'TestAssembly', new object[] { 'a', 12 }, typeof(ITest));

// if the created instance is not subclass of the base type,
// exception will be thrown
try {
      cm.CreateInstance('ns1', 'AnotherClass', typeof(ITest));
      fail();
} catch (InstanceCreationException ice) {
      // good
}

7. Load the configuration values from pluggable datasource.
// A sample implementation of IDataHandler is provided to
// load/save namespaces from database, which will be instantiated
// by PluggableHandler dynamically from the mxml file.
// the ddl for the namespaces, properties, and values table should
// likes follows:
namespaceTable (
    id LONG PRIMARY KEY,
```

```
    name VARCHAR (255)
);

propertyTable (
    id LONG PRIMARY KEY,
    namespace_id LONG,
    name VARCHAR (255)
);

valueTable (
      id LONG PRIMARY KEY,
      prop_id LONG,
      value VARCHAR (255)
);

class DatabaseHandler : IDataHandler {
      public const string ConnectionUrl = "connectionUrl";

      private string namespaceTable = null;
      private string propertyTable = null;
      private string valueTable = null;

      private SqlConnection conn = null;

      public DatabaseHandler(IDictionary props) {
            conn =
              new SqlConnection((string) props[ConnectionUrl]);
            namespaceTable = (string) props["namespaceTable"];
            propertyTable = (string) props["propertyTable"];
            valueTable = (string) props["valueTable"];
      }

      public IList Load(ErrorCallback errorCallback) {
            IList list = new ArrayList();

            conn.Open();
            SqlCommand cmd = new SqlCommand();
            cmd.CommandText = "select * from " + namespaceTable;
            cmd.Connection = conn;
            SqlDataReader reader = cmd.ExecuteQuery();
            // get all the namespaces in the database
            IDictionary map = new HashTable();
            while (reader.Read()) {
                  long nsId = reader.GetLong(0);
                  string nsName = reader.GetString(1);
                  map[nsId] = new Namespace(nsName);
            }
            reader.Close();

            // select all the properties for each namespace
            // and retrieve the values for these properties too.

            // finally return the loaded namespaces
            new ArrayList(map.Values);
      }
```

```
        public void Save(IList namespaces) {
                // saving the namespaces to database
                // the code will be similar as above
        }
}


// Here is the corresponding mxml file that will be loaded by
// the ConfigManager, which is finally parsed by the
// PluggableHandler to create the DatabaseHandler instance.
<?xml version="1.0" ?>
<PluggableHandler>
        <Class>TopCoder.Util.ConfigManager.DatabaseHandler</Class>
        <!-- assembly is optional -->
        <Assembly>xxx</Assembly>
        <Properties>
                <Property name="connectionUrl">
                        <Value>xx</Value>
                </Property>
                <Property name=”namespaceTable”>
                        <Value>namespaces</Value>
                </Property>
                <Property name=”propertyTable”>
                        <Value>properties</Value>
                </Property>
                <Property name=”valueTable”>
                        <Value>values</Value>
                </Property>
        </Properties>
</PluggableHandler>


// add the mxml file into ConfigManager
cm.LoadFile(“database.mxml”);
string value = cm.GetValue(“ns_in_database”, “prop”);
```

5. Future Enhancements

More implementation of IDataHandler interface can be added to load the configuration values from pluggable different data sources. And the client may choose to implement the IParsingPreference and IStringConverter