Object Factory Configuration API Plugin 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**.

The Object Factory component provides a generic infrastructure for dynamic object creation at run-time. It provides a standard interface to create objects based on some specifications. This component provides one such specification using the *ConfigurationObject* interface from the Configuration API component.

This component provides an implementation of the *SpecificationFactory* interface from Object Factory component and uses *ConfigurationObject* to supply the specifications.

The hierarchical structure of the *ConfigurationObject* is compatible with the *ConfigManager* configuration settings, so that existing configuration files that are used for *ConfigManagerSpecificationFactory* (from Object Factory Config Manager Plugin component) can be used by the combination of Configuration Persistence and *ConfigurationObjects* with no or little change.

An enterprise application requires dynamic object creation based on configuration settings, where the configuration settings are managed by the *ConfigurationObjects* from the Configuration API component. This component is used for that purpose.

This component is intended to be used as a plugin for Object Factory component so please read the Component Specification from Object Factory component.

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.
- Used the latest versions of dependency components.
- Some trivial source code changes are performed to make the component meet TopCoder standards.

1.1 Design Patterns

Strategy pattern:

- Not directly in this design: ObjectFactory uses different implementations of the *SpecificationFactory* instance.

1.2 Industry Standards

- None.

1.3 Required Algorithms

Considerations regarding the hierarchy structure

This component is intended to re-use the same structure used in the SpecificationFactory created using Configuration Manager component (defined in Object Factory Config

Manager Plugin component). The goal here is to provide nested children available from Configuration Manager (but here we don't have this "nested" functionality), but this can be easy achieved using the "children" functionality provided by Configuration API (that are other ConfigurationObject instances linked in a parent child relationship).

For the next configuration defined in an XML and loaded until now in Object Factory with Configuration Manager:

```
 <Property name="frac:default">
   <Property name="type">
     <Value>com.topcoder.util.objectfactory.testclasses.TestClass</Value>
   </Property>
</Property>
```

The "type" here is nested by "name" property. Using the "." notation for a better reading results the "name.type" nested property.

To show the hierarchy structure in this way we will do the following:

- Non-nested properties will be expressed using ConfigurationObject properties
- Nested properties will be expressed using child ConfigurationObject instances (each nested property will be a child ConfigurationObject instance).

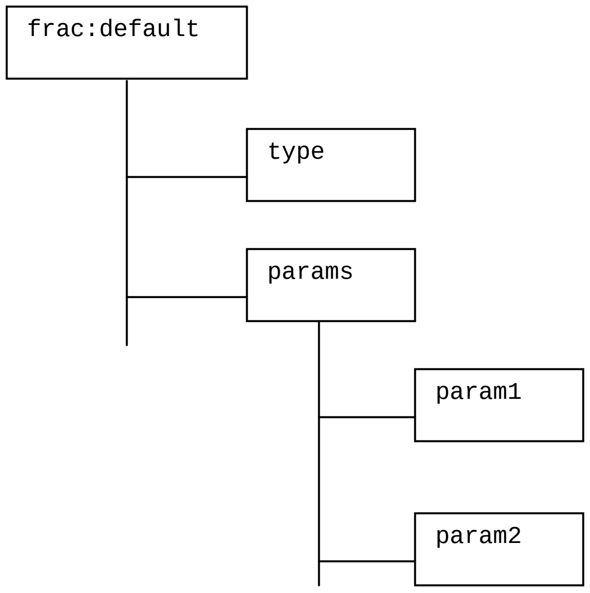
To better show this we will use the next configuration defined in an XML and loaded until now in Object Factory with Configuration Manager:

```
<Property name="frac:default">
  <Property name="type">
    <Value>com.topcoder.util.objectfactory.testclasses.TestClass</Value>
  </Property>
  <Property name="params">
    <Property name="param1">
      <Property name="type">
        <Value>int</Value>
      </Property>
      <Property name="value">
        <Value>1</Value>
      </Property>
    </Property>
    <Property name="param2">
      <Property name="type">
        <Value>String</Value>
      </Property>
      <Property name="value">
        <Value>Strong</Value>
      </Property>
    </Property>
  </Property>
</Property>
```

The result is as follows:

```
frac:default.type = com.topcoder.util.objectfactory.testclasses.TestClass
frac:default.params.param1.type = int
frac:default.params.param1.value = 1
frac:default.params.param2.type = String
frac:default.params.param2.value = Strong
```

And the hierarchy structure is:



So we have 3 ConfigurationObjects.

For the arrays we need one ConfigurationObject because we have just only one level:

```
<Property name="intArray">
       <Property name="arrayType">
              <Value>int</Value>
       </Property>
       <Property name="dimension">
              <Value>2</Value>
       </Property>
       <Property name="values">
              <Value> {{1, 2}, {3, 4}} </Value>
       </Property>

</Property>
```

The result is:

```
intArray.arrayType=int
intArray.dimension=2
intArray.values={{1,2},{3,4}}
```

Note: *The algorithms are the same with the algorithms from Object Factory component and the configurations are represented like for the Configuration Manager configurations because the ConfigurationObject hierarchy should be compatible with Configuration Manager configuration settings. Also configurations can be placed in an XML file in this format compatible with Configuration Manager and using Configuration Persistence component and Configuration API those configurations from XML file cam be retrieved without using Configuration Manager (and the intention of TopCoder is to move from*

Configuration Manager to Configuration API). Another reason that configurations are presented like for Configuration Manager is because there are a lot of components that are dependent with Configuration Manager and their configurations are presented in this manner (XML style).

This section will show two algorithms:

- How the custom, *ConfigurationObject*-backed specification factory reads the configuration information to create *ObjectSpecifications*, and how it makes sure the definitions are not cyclical.
- How the specification factory matches the passed type and identifier to a specification in its store of specifications.

The developer is encouraged to improve on these algorithms.

1.3.1 Creating ObjectSpecifications from configuration

This section details how the *ConfigurationObjectSpecificationFactory* reads configuration information and creates *ObjectSpecifications*.

A typical configuration contains several top-level children for the *names* it supports, so the root ConfigurationObject is a container for top-level children. Each *name* of the child is actually a concatenation of the *key* and a modifying *identifier*. Each such child will contain nested children for *type*, *jar*, and optional *params*. The *params* child will contain one or more *param<N>* (<N> is a 1..n, so the parameters are ordered) children that will contain a simple type, a complex type, or a null. The simple type child will have two nested children: *type* and *value*. The complex child will have one nested child – *name* – that maps to another top-level child. Null values for Objects (all complex types plus String) will have one nested child: the afore-mentioned *type.* As such, it is valid to look at a null value as a simple type, but it will be treated as a separate type.

One approach is to first create an *ObjectSpecification* for each top-level child, with the *name* being split into the key and the identifier, and to create *ObjectSpecification* parameters for the simple parameters. Once this is done, then the second step would be to link the complex parameters together, as it is a requirement that a reference complex parameter must be defined, or an *IllegalReferenceException* will be thrown. Every top-level specification will be then added to a map with the *key* as a key. The specification will be added to a List because there can be multiple entries with the same *key*.

In general, if there’s any problem during this process, an *IllegalReferenceException* will be thrown. Note that if there’s a problem with accessing the configuration, which is a separate issue, then a *SpecificationConfigurationException* will be thrown.

The detailed configuration specification can be found in section 3.2, including which types are considered “simple.”

Working with the example from the introduction, which samples most of the permutations:

```
new com.Frac(2, "Strong", bar);
where bar = new com.Bar(2.5F, new StringBuffer());
```

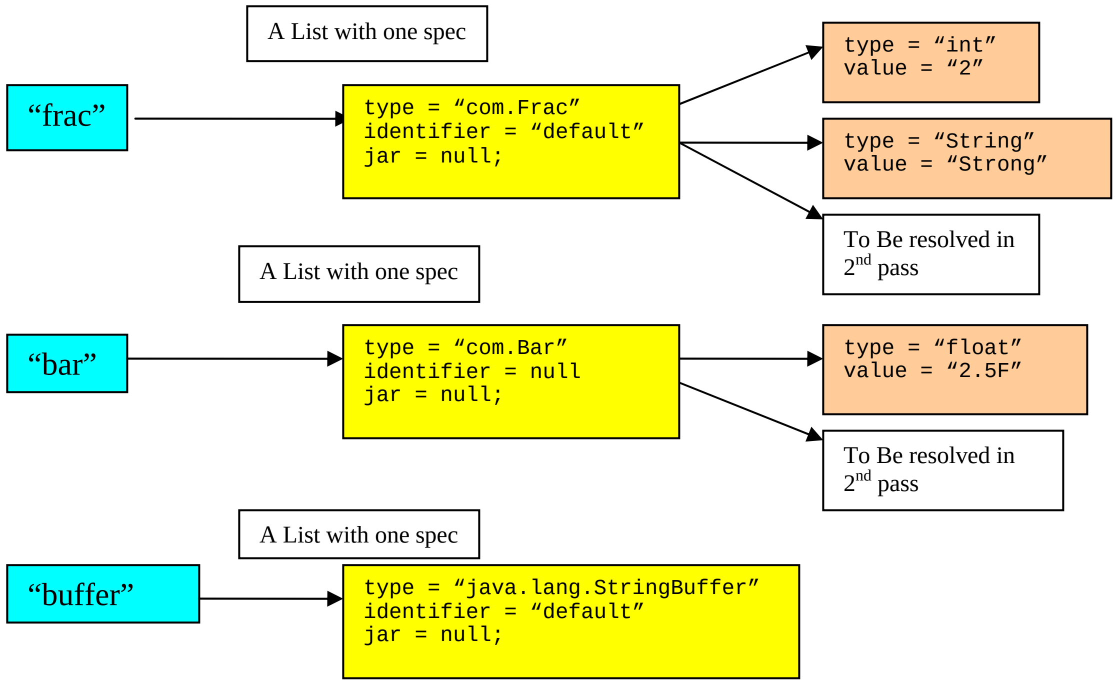
The configuration would look like the following (the file being stripped to the properties):

```
<Property name="frac:default">
```

```
        <Property name="type">
              <Value>com.Frac</Value>
        </Property>
        <Property name="params">
              <Property name="param1">
                    <Property name="type">
                          <Value>int</Value>
                    </Property>
                    <Property name="value">
                          <Value>2</Value>
                    </Property>
              </Property>
              <Property name="param2">
                    <Property name="type">
                          <Value>String</Value>
                    </Property>
                    <Property name="value">
                          <Value>Strong</Value>
                    </Property>
              </Property>
              <Property name="param3">
                    <Property name="name">
                          <Value>bar</Value>
                    </Property>
              </Property>
        </Property>
  </Property>

  <Property name="bar">
        <Property name="type">
              <Value>com.Bar</Value>
        </Property>
        <Property name="params">
              <Property name="param1">
                    <Property name="type">
                          <Value>float</Value>
                    </Property>
                    <Property name="value">
                          <Value>2.5F</Value>
                    </Property>
              </Property>
              <Property name="param2">
                    <Property name="name">
                          <Value>buffer:default</Value>
                    </Property>
              </Property>
        </Property>
  </Property>

  <Property name="buffer:default">
        <Property name="type">
              <Value>java.lang.StringBuffer</Value>
        </Property>
  </Property>
```

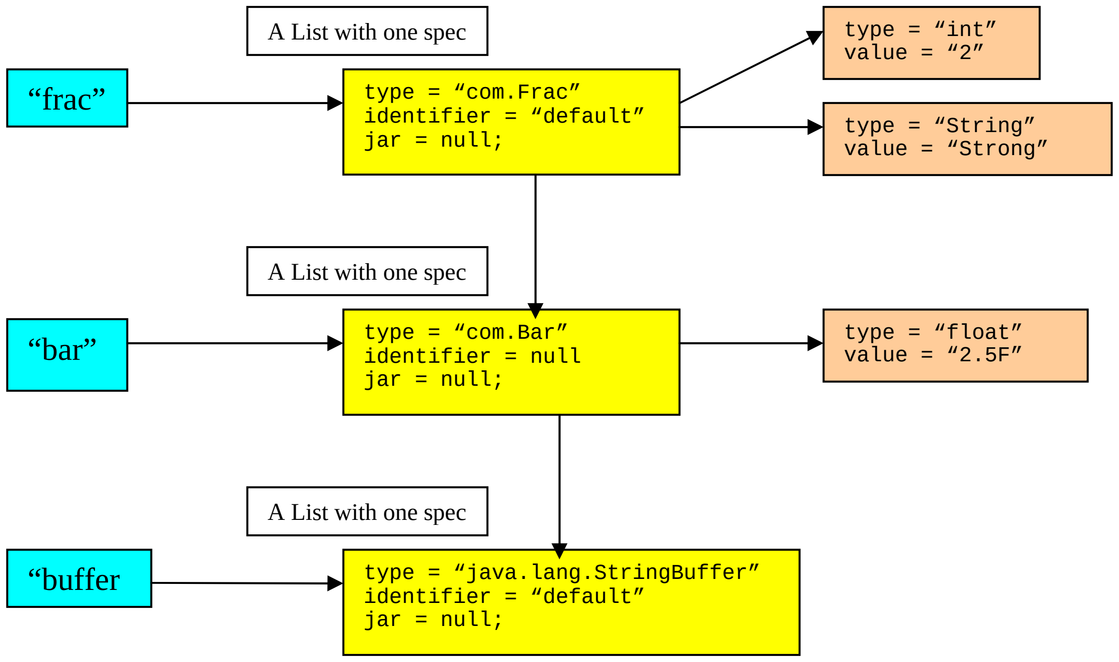
Following the algorithm, the first pass would result in 3 top-level entries in the map, with each entry being a List with one *ObjectSpecification* (Illustration 1), which contains the simple *ObjectSpecification* parameters:

Illustration 1: First Pass



The second pass will simply involve resolving the complex parameters to *ObjectSpecification* entries in the map (Illustration 2):

Illustration 2: Second Pass



The above example can be also used to illustrate how null values are stated:

```
new com.Frac(2, "Strong", null);
```

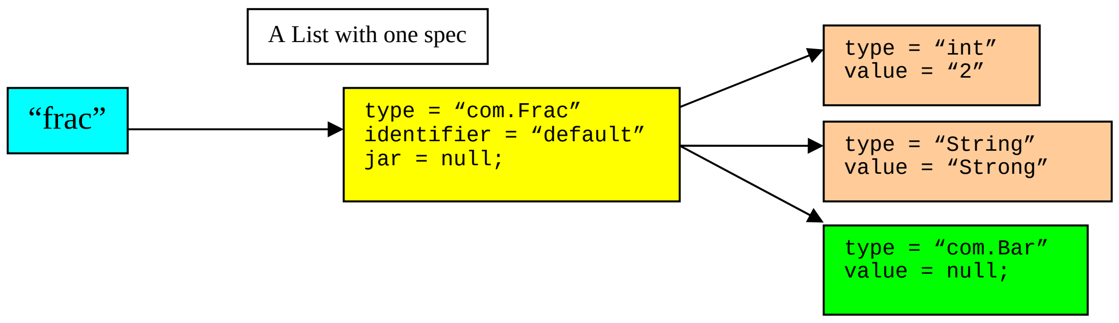
The null represents a null Bar object. The above configuration would change to the following:

```
<Property name="frac:nullbar">
	<Property name="type">
		<Value>com.Frac</Value>
	</Property>
	<Property name="params">
		<Property name="param1">
			<Property name="type">
				<Value>int</Value>
			</Property>
			<Property name="value">
				<Value>2</Value>
			</Property>
		</Property>
		<Property name="param2">
			<Property name="type">
				<Value>String</Value>
			</Property>
			<Property name="value">
				<Value>Strong</Value>
			</Property>
		</Property>
		<Property name="param3">
			<Property name="type">
				<Value>com.Bar</Value>
			</Property>
		</Property>
	</Property>
</Property>
```

The singular presence of the type nested child for param3 instructs the factory to treat this null of specified type.

When such a null is encountered, it will be saved as an *ObjectSpecification* with the entered type and value of null (similarly to a simple type). The result would be a modified version of Illustration 2:

Illustration 3: Object with a null specification



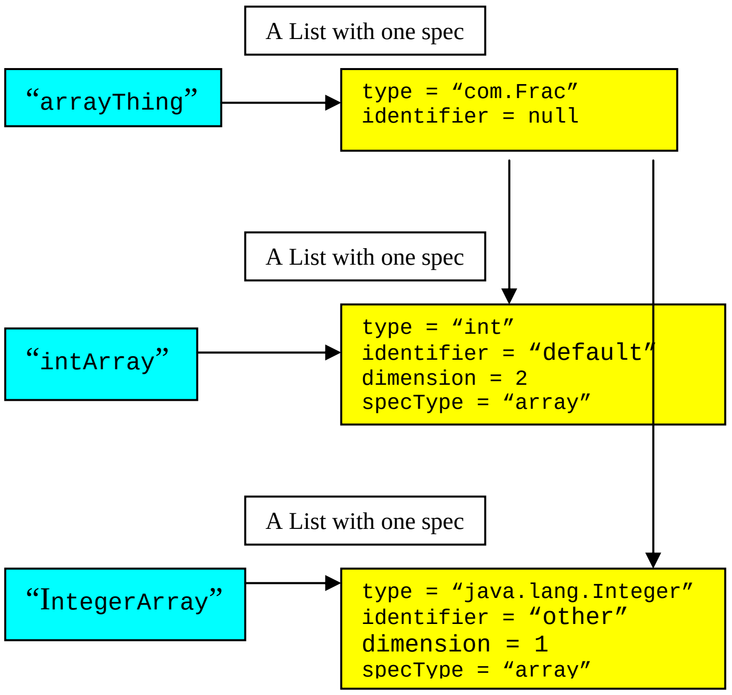
1.3.2 *Dealing with Arrays in configuration*

This design treats arrays as a special type of complex type, and the configuration will reflect this, by having different sub-children under the top-level child: *arrayType*, *dimension*, and *values*. The *arrayType* child will hold the type of the array (like int, String, java.net.URL), the *dimension* child will hold the dimension of the array (1, 2, etc), and *values* child will hold the values of the array, with braces delimiting dimensions, and a comma delimiting the values in a dimension. If the type is a simple type (see Table 2 in section 3.2), then the *values* will contain actual values. Otherwise, the values will be names to other complex entries in the specification. To illustrate, there follows a specification for a type with two arrays, one of a simple type, and one of a complex type

```
<Property name="arrayThing">
        <Property name="type">
                <Value>com.Frac</Value>
        </Property>
        <Property name="params">
                <Property name="param1">
                        <Property name="name">
                                <Value>intArray</Value>
                        </Property>
                </Property>

                <Property name="param2">
                        <Property name="name">
                                <Value>IntegerArray</Value>
                        </Property>
                </Property>

        </Property>
</Property>

<Property name="intArray">
        <Property name="arrayType">
                <Value>int</Value>
        </Property>
        <Property name="dimension">
                <Value>2</Value>
        </Property>
        <Property name="values">
                <Value>{{1,2},{3,4}}</Value>
        </Property>
</Property>

<Property name="IntegerArray">
        <Property name="arrayType">
                <Value>Integer</Value>
        </Property>
        <Property name="dimension">
                <Value>1</Value>
        </Property>
        <Property name="values">
                <Value>{integer:default, integer:other}</Value>
        </Property>
</Property>

… other entries (integer:default, integer:other) omitted for clarity
```

Once the algorithm in section 1.3.2 is done, the map should look like in illustration 3 (with most entries, like jar, removed for brevity):

Illustration 3: Mapping of an array



Not shown in Illustration 3, the simple specifications for the first, 2-dimensional int array would be held in the *params* member of the *ObjectSpecification* as an Object[][] of simple *ObjectSpecifications*, and the 1-dimensional Integer array would be held as a Object[] array. For the developer, one way to create arbitrary-dimension arrays is to use the *Array* class in java.lang.reflect package, cast the instance to Object[] to set the *params* part of the *ObjectSpecification*, and fill each value in the array with simple *ObjectSpecifications*.

If null values are desired, then the “null” keyword would have to be used, but only for Objects, since null primitives are not allowed. This excerpt demonstrates

```
<Property name="IntegerArray:nullable">
	<Property name="arrayType">
		<Value>Integer</Value>
	</Property>
	<Property name="dimension">
		<Value>1</Value>
	</Property>
	<Property name="values">
		<Value>{integer:default, null}</Value>
	</Property>
```

```
</Property>
```

1.3.3 *Analyzing the specifications for cyclical definitions*

This component checks for cyclical definitions. Simply put, it will make sure that an object does not point directly or indirectly to itself. This is performed once all specifications are resolved, and involves a simple depth-first search for any multiple use of the same specification in a branch. If this is encountered, then a *SpecificationConfigurationException* will be thrown.

1.3.4 *Obtaining ObjectSpecifications from ConfigurationObjectSpecificationFactory*

As shown in section 1.3.1, the *ConfigurationObjectSpecificationFactory* stores its specifications in a List in a map with the type as the key. When calling the specification factory, the caller passes a required key String or type Class, and an optional identifier.

If both parameters are passed, the specification factory will obtain the specification List that is mapped to the key/type, and will run through the List to find the first specification that matches the passed identifier. If no identifier was passed, then it will match the first specification in the List not to have an identifier. As such, having no identifier can be considered to be also an identifier.

If there is no match, then *UnknownReferenceException* will be thrown.

1.4 Component Class Overview

ConfigurationObjectSpecificationFactory

This class is a concrete implementation of the *SpecificationFactory* backed by the *ConfigurationObject* (from Configuration API component) as the source of the configuration. All specifications are loaded on startup, and placed in a map. When called, the map will be queried for the specification that matches the requested type and identifier.

1.5 Component Exception Definitions

None

1.6 Thread Safety

This component is thread safe because its main class is thread safe from the following reasons: it has no mutable state (its member it is initialized in constructor and never changed afterwards). It's taken into account that ObjectFactory uses mutable ObjectSpecification instances in thread safe manner.

An issue regarding thread safety is related to ConfigurationObject that could be modified externally while reading the configurations. Even that it is highly unlikely that the ConfigurationObject will be shared/modified while the particular SpecificationFactory implementation is being constructed, the user should prevent this scenario to ensure thread safety.

The thread safety should be considered with *ObjectFactory* class and with Object Factory component so users are strongly advised to read thread safety section from Object Factory component.

If ObjectFactory will be used in multi-threaded environments, its plugins must be thread safe and this component ensure this requirement.

Thread safety of the 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

- **Object Factory 2.2** – Object Factory Configuration API Plugin 1.1 is a plugin for Object Factory component. *ObjectFactory* objects will be created using the new implementation of *SpecificationFactory* that uses Configuration API for configurations.
- **Configuration API 1.1** – Object Factory Configuration API Plugin 1.1 uses *ConfigurationObject* from Configuration API for configurations in implementation of *SpecificationFactory*.

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.

3. Installation and Configuration

3.1 Package Name

com.topcoder.util.objectfactory.impl

3.2 Configuration Parameters

The top-level children will be the names of the supported objects. Each name will be the concatenation of the key, a semi-colon, and an identifier. If there is no identifier, then the key would be by itself:

Table 1: Object specification configuration

Parameter	Description	Values
`<name>`	Either <key>:<identifier> or <key>. Key is required, identifier is optional. The key can be the same as the type.	Any
`<name>.jar`	Valid reference to a JAR file.	Valid jar name. Optional.
`<name>.type`	Fully-qualified name of the class to be instantiated.	Valid type. Required.
`<name>.arrayType`	For array types. The type of array.	Required if array type.
`<name>.dimension`	For array types. The dimension of the array.	1..n. Required if array type.
`<name>.values`	For array types. The actual brace-delimited values of the array.	{{1,2},{2,3}}. Required if array type.
`<name>.params`	Container property for parameters to pass to the constructor.	N/A
`<name>.params.param<n>`	<n> will be 1..n, so the parameters are ordered as per desired constructor.	
`<name> params.param<n>.type`	One of the eight primitives, or just a "String" (see Table 2 below), for simple types, or an Object if values is null.	"int","String". Required if is a simple type or null Object. If null, then must be an Object (not a primitive).
`<name>.params.param<n>.value`	Value of the simple type.	"2" "true". Required if is a simple type. Must be absent if null value is desired.
`<name>.params.param<n>.name`	The same specification as the <name> entry.	Must be an existing <name>.

Table 2: Recognized simple types (8 primitive types and String)

Recongized simple type	Maps to the following class
`"int"`	java.lang.Integer
`"long"`	java.lang.Long
`"short"`	java.lang.Short
`"byte"`	java.lang.Byte
`"char"`	java.lang.Character
`"float"`	java.lang.Float
`"double"`	java.lang.Double
`"boolean"`	java.lang.Boolean
`"String" or "java.lang.String"`	java.lang.String

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

No special steps are required to use this component. The jars for the Object Factory and Configuration API components must be added, as well as any extra jars that will be used in the course of creating objects.

4.3 Demo

This section shows the usage of this component. We define two short demos:

4.3.1 Setup

For the purposes of the demo, we consider the next ***config.xml*** file:

```
<CMConfig>
	<Config name="valid_config">
		<Property name="frac:default">
			<Property name="type">

	<Value>com.topcoder.util.objectfactory.impl.TestClass</Value>
			</Property>
			<Property name="params">
				<Property name="param1">
					<Property name="type">
						<Value>int</Value>
					</Property>
					<Property name="value">
						<Value>2</Value>
					</Property>
				</Property>
				<Property name="param2">
					<Property name="type">
						<Value>String</Value>
					</Property>
					<Property name="value">
						<Value>Strong</Value>
					</Property>
				</Property>
				<Property name="param3">
					<Property name="name">
						<Value>bar</Value>
					</Property>
				</Property>
			</Property>
		</Property>

		<Property name="bar">
			<Property name="type">
				<Value>com.topcoder.util.objectfactory.impl.Bar</Value>
			</Property>
			<Property name="params">
				<Property name="param1">
					<Property name="type">
						<Value>float</Value>
					</Property>
					<Property name="value">
						<Value>2.5F</Value>
					</Property>
```

```
                          </Property>
                          <Property name="param2">
                                <Property name="name">
                                      <Value>buffer:default</Value>
                                </Property>
                          </Property>
                    </Property>
             </Property>

             <Property name="buffer:default">
                    <Property name="type">
                          <Value>java.lang.StringBuffer</Value>
                    </Property>
             </Property>

      </Config>
</CMConfig>
```

4.3.2 Create the needed object using ObjectFactory created from a ConfigurationObject (built manually)

```
public void testDemo1() throws Exception {
        // the root ConfigurationObject
        ConfigurationObject root = new DefaultConfigurationObject("root");

        /*
         * create the TestClass configuration object
         */
        // specifying the identifier and the key
        ConfigurationObject testClassObject = new
DefaultConfigurationObject("frac:default");
        // the type of the object to be created
        testClassObject.setPropertyValue("type",
"com.topcoder.util.objectfactory.impl.TestClass");

        // create the params configuration object
        ConfigurationObject params = new DefaultConfigurationObject("params");

        // create one parameter
        ConfigurationObject param1 = new DefaultConfigurationObject("param1");
        param1.setPropertyValue("type", "int");
        param1.setPropertyValue("value", "1");
        // create another parameter
        ConfigurationObject param2 = new DefaultConfigurationObject("param2");
        param2.setPropertyValue("type", "String");
        param2.setPropertyValue("value", "Strong");
        // create another parameter
        ConfigurationObject param3 = new DefaultConfigurationObject("param3");
        param3.setPropertyValue("name", "bar");

        // add the children
        params.addChild(param1);
        params.addChild(param2);
        params.addChild(param3);
        testClassObject.addChild(params);

        /*
         * create the Bar configuration object
         */
        // specifying the identifier and the key
        ConfigurationObject barObject = new DefaultConfigurationObject("bar");
        // the type of the object to be created
        barObject.setPropertyValue("type",
"com.topcoder.util.objectfactory.impl.Bar");

        // create the params configuration object
        params = new DefaultConfigurationObject("params");

        // create one parameter
        param1 = new DefaultConfigurationObject("param1");
        param1.setPropertyValue("type", "float");
        param1.setPropertyValue("value", "100.0");
        // create another parameter
        param2 = new DefaultConfigurationObject("param2");
        param2.setPropertyValue("name", "buffer");

        // add the children
        params.addChild(param1);
        params.addChild(param2);
        barObject.addChild(params);

        /*
         * create the StringBuffer configuration object
         */
        // specifying the identifier and the key
        ConfigurationObject bufferObject = new
DefaultConfigurationObject("buffer");
        bufferObject.setPropertyValue("type", "java.lang.StringBuffer");

        // create the params configuration object
        params = new DefaultConfigurationObject("params");

        // create one parameter
        param1 = new DefaultConfigurationObject("param1");
        param1.setPropertyValue("type", "String");
        param1.setPropertyValue("value", "string buffer");

        // add the children
        params.addChild(param1);
        bufferObject.addChild(params);

        // add to the root
        root.addChild(testClassObject);
        root.addChild(barObject);
        root.addChild(bufferObject);

        // create ConfigurationObjectSpecificationFactory
        // from the given configuration object
        ConfigurationObjectSpecificationFactory cosf = new
ConfigurationObjectSpecificationFactory(root);

        //create the object factory
        ObjectFactory objFactory = new ObjectFactory(cosf);

        //create the needed StringBuffer object using the object factory
        StringBuffer buffer = (StringBuffer) objFactory.createObject("buffer");

        //create the needed Bar object using the object factory
        Bar bar = (Bar) objFactory.createObject("bar");

        //create the needed TestClass object using the object factory
        TestClass testClass = (TestClass) objFactory.createObject("frac",
"default");
}
```

4.3.3 Create the needed object using ObjectFactory created from a ConfigurationObject (built using Configuration Persistence component and configurations stored in a xml configuration file like presented in config.xml from above)

```
public void testDemo2() throws Exception {
        // build the XMLFilePersistence
        XMLFilePersistence xmlFilePersistence = new XMLFilePersistence();

        //load the ConfigurationObject from the input file
        ConfigurationObject cfgObject = xmlFilePersistence.loadFile("test", new
File(TEST_FILES + "config.xml"));

        // create ConfigurationObjectSpecificationFactory
        // from the given configuration object
        ConfigurationObjectSpecificationFactory cosf =
            new
ConfigurationObjectSpecificationFactory(cfgObject.getChild("valid_config"));

        //create the object factory
        ObjectFactory objFactory = new ObjectFactory(cosf);

        //create the needed Bar object using the object factory
        Bar bar = (Bar) objFactory.createObject("bar");

        //create the needed TestClass object using the object factory
        TestClass testClass = (TestClass) objFactory.createObject("frac",
"default");
}
```

Note: *this component is a plugin for Object Factory component so for full details of the operations available for Object Factory please see demo from Object Factory component.*

5. Future Enhancements

- None at this moment.