Object Factory Config Manager 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 configuration settings or some other specifications. This component provides a ConfigManager-backed specification factory plugin.

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 – this component defines an implementation of SpecificationFactory that can be plugged into ObjectFactory that serves as a strategy context in this case.

1.2 Industry Standards

None

1.3 Required Algorithms

This section will show two algorithms:

- How the custom, ConfigManager-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.

1.3.1 Creating ObjectSpecifications from configuration

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

A typical configuration contains several top-level properties for the *names* it supports. Each *name* is actually a concatenation of the *key* and a modifying *identifier*. Each such property will contain sub properties for *type*, *jar*, and optional *params*. The *params* property will contain one or more *param<N>* (<N> is a 1..n, so the parameters are ordered) properties that will contain a simple type, a complex type, or a null. The simple type property will have two sub-properties: *type* and *value*. The complex property will have one sub-property – *name* – that maps to another top-level property. Null values for Objects (all complex types plus String) will have one sub-property: 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 property, 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 *ConfigurationException* 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());
```

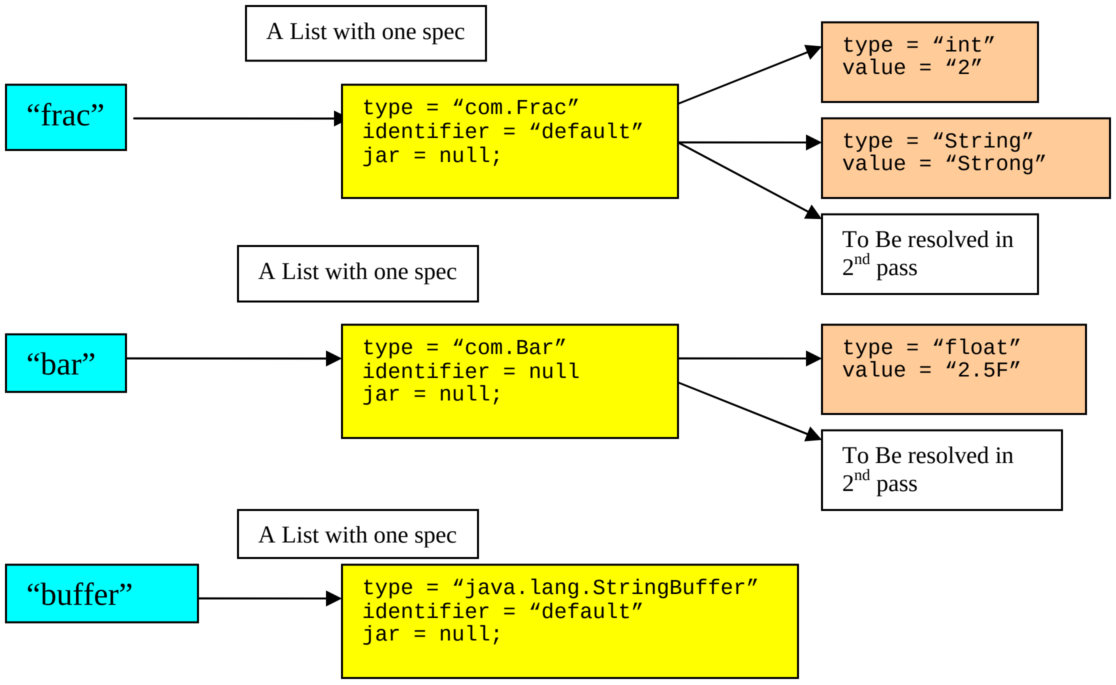
The configuration would look like the following (the file being stripped to the properties):

```
<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>
```

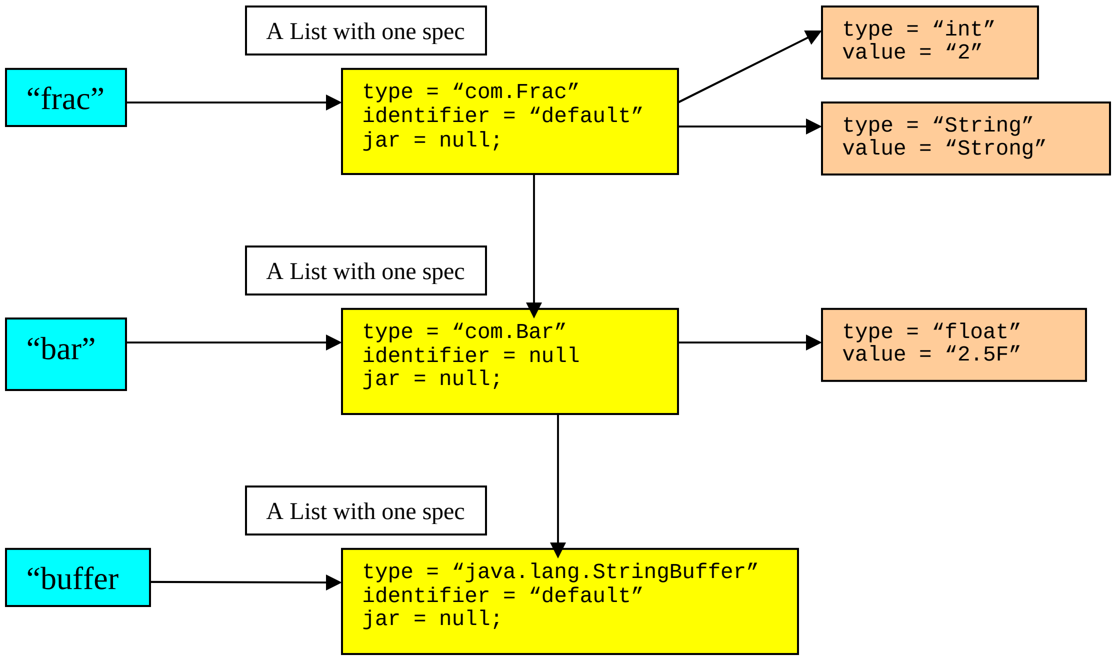
Following the algorithm, the first pass would result in 3 top-level entries in the map, with each entry being a List with one *ObjectSpecification* (Illustration 1), which contains the simple *ObjectSpecification* parameters:

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);
```

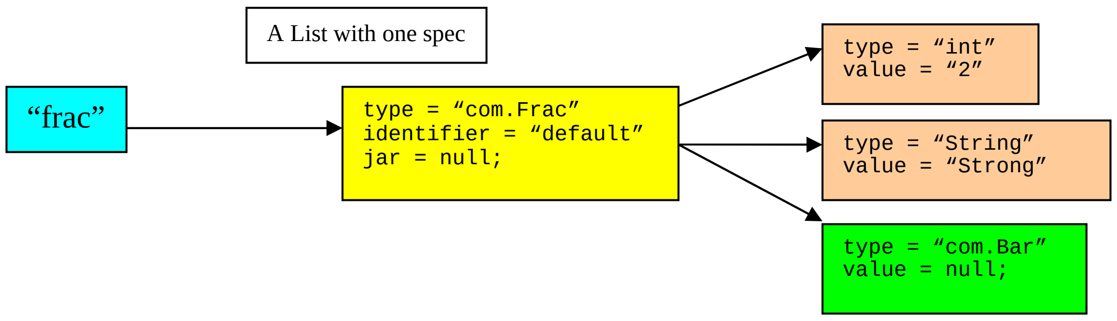
The null represents a null Bar object. The above configuration would change to the following:

```
<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 sub-property for param3 instructs the factory to treat this as a 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



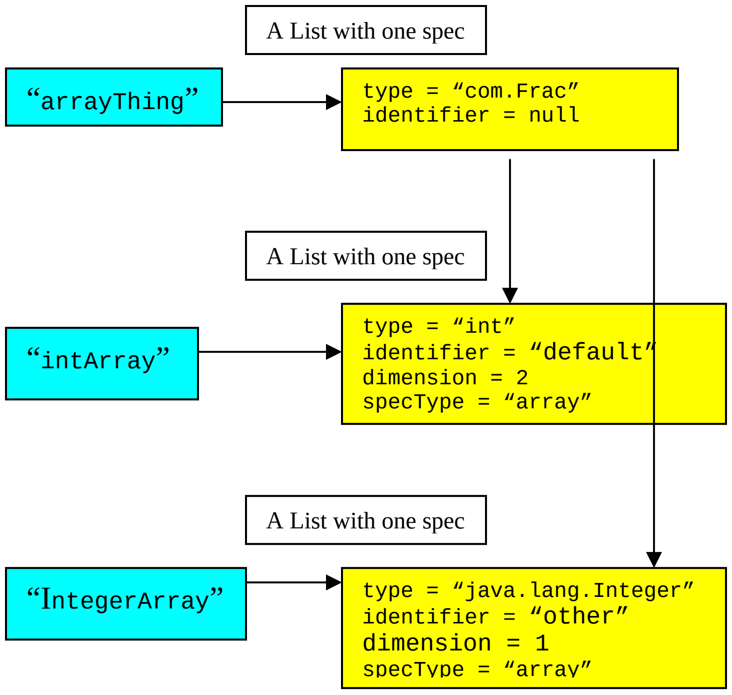
1.3.2 *Dealing with Arrays in configuration*

This design treats arrays as a special type of complex type, and the configuration will reflect this, by having different sub-properties under the top-level property: *arrayType*, *dimension*, and *values*. The *arrayType* property will hold the type of the array (like int, String, java.net.URL), the *dimension* property will hold the dimension of the array (1, 2, etc), and *values* property 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 first-depth 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 ConfigManagerSpecificationFactory*

As shown in section 1.3.1, the *ConfigManagerSpecificationFactory* 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

ConfigManagerSpecificationFactory:
Concrete implementation of the *SpecificationFactory* backed by the *ConfigManager* as the source of the configuration. All specifications are loaded on startup.
Changes in 1.1:
- Made all private methods static.
- Specified generic parameters for all generic types in the code.

1.5 Component Exception Definitions

None

1.6 Thread Safety

This component is thread-safe. The specifications are read during object construction, after which the object is immutable. As such, the fact that the internal maps and lists are of the non-thread-safe flavor, the class immutability ensures thread-safety.

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**

    - Defines SpecificationFactory interface implemented by *ConfigManagerSpecificationFactory* in this component.

- **Configuration Manager 2.2.1**
    - Used to specify object definitions in the *ConfigManagerSpecificationFactory*.

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 properties 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 were 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 params 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)

Recognized 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.

4.3 Demo

4.3.1 Setup

For the purposes of the demo, we can start with the specification in section 1.3.1 that defines the following classes:

```
<Config name="valid_config">
  <Property name="frac:default">
    <Property name="type">
      <Value>com.topcoder.util.objectfactory.testclasses.TestClass1</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>
  </Property>
  <Property name="int:default">
    <Property name="type">
      <Value>com.topcoder.util.objectfactory.testclasses.TestClass1</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>
  </Property>
```

```
<Property name="frac1:default">
  <Property name="type">
    <Value>com.topcoder.util.objectfactory.testclasses.TestClass1</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>com.topcoder.util.objectfactory.testclasses.TestClass2</Value>
      </Property>
    </Property>
  </Property>
</Property>
<Property name="frac1">
  <Property name="type">
    <Value>com.topcoder.util.objectfactory.testclasses.TestClass1</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>com.topcoder.util.objectfactory.testclasses.TestClass2</Value>
      </Property>
    </Property>
  </Property>
</Property>
<Property name="bar">
  <Property name="type">
    <Value>com.topcoder.util.objectfactory.testclasses.TestClass2</Value>
  </Property>
  <Property name="params">
    <Property name="param1">
      <Property name="name">
        <Value>frac:default</Value>
      </Property>
    </Property>
    <Property name="param2">
      <Property name="type">
        <Value>float</Value>
      </Property>
      <Property name="value">
        <Value>2.5F</Value>
      </Property>
    </Property>
  </Property>
</Property>
<Property name="buffer:default">
  <Property name="type">
    <Value>java.lang.StringBuffer</Value>
  </Property>
</Property>
<Property name="intArray:arrays">
  <Property name="arrayType">
    <Value>int</Value>
  </Property>
  <Property name="dimension">
    <Value>2</Value>
```

```
      </Property>
      <Property name="values">

<Value>{{1,2},{3,4},{3,4},{3,4},{3,4},{3,4},{3,4},{3,4},{3,4},{3,4},{3,4},{3,4},{3
,4},{3,4},{3,4},{3,4},{3,4}}</Value>
      </Property>
    </Property>
    <Property name="hashset">
      <Property name="type">
        <Value>java.util.HashSet</Value>
      </Property>
    </Property>
    <Property name="test:arraylist">
      <Property name="type">
        <Value>java.util.ArrayList</Value>
      </Property>
      <Property name="params">
        <Property name="param1">
          <Property name="type">
            <Value>int</Value>
          </Property>
          <Property name="value">
            <Value>4</Value>
          </Property>
        </Property>
      </Property>
    </Property>
    <Property name="test:collection">
      <Property name="arrayType">
        <Value>java.util.Collection</Value>
      </Property>
      <Property name="dimension">
        <Value>1</Value>
      </Property>
      <Property name="values">
        <Value>{hashset, null, test:arraylist}</Value>
      </Property>
    </Property>
    <Property name="int:collection">
      <Property name="arrayType">
        <Value>java.util.Collection</Value>
      </Property>
      <Property name="dimension">
        <Value>1</Value>
      </Property>
      <Property name="values">
        <Value>{hashset, null, test:arraylist}</Value>
      </Property>
    </Property>
    <Property name="testcollection">
      <Property name="arrayType">
        <Value>java.util.Collection</Value>
      </Property>
      <Property name="dimension">
        <Value>1</Value>
      </Property>
      <Property name="values">
        <Value>{hashset, null, test:arraylist}</Value>
      </Property>
    </Property>
    <Property name="objectArray">
      <Property name="arrayType">
        <Value>java.lang.Object</Value>
      </Property>
      <Property name="dimension">
        <Value>3</Value>
      </Property>
      <Property name="values">
        <Value>{{{frac:default, bar, null}}}</Value>
      </Property>
    </Property>
    <Property name="int:Mismatch">
      <Property name="arrayType">
        <Value>java.util.Collection</Value>
      </Property>
      <Property name="dimension">
        <Value>1</Value>
      </Property>
      <Property name="values">
        <Value>{hashset, objectArray}</Value>
      </Property>
    </Property>
    <Property name="typeMismatch">
      <Property name="arrayType">
        <Value>java.util.Collection</Value>
      </Property>
      <Property name="dimension">
        <Value>1</Value>
      </Property>
      <Property name="values">
        <Value>{hashset, objectArray}</Value>
      </Property>
    </Property>
  </Config>
```

4.3.2 ObjectSpecification retrieval demo

This section details the use of the single method in this component: to retrieve object specifications.

```
// instantiate factory with specification factory
SpecificationFactory factory = new
ConfigManagerSpecificationFactory("valid_config");

// obtain the configured default frac
ObjectSpecification spec = factory.getObjectSpecification("frac",
"default");
```

5. Future Enhancements

None at this time