Object Factory 2.0 Component Specification

1. Design

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. Using an object factory facilitates designing a broader solution by allowing the specific details of the instantiated class to be designed at a later time.

The purpose of this next major version of the Object Factory is to provide additional functionality:

- Configure an object definition, which can be called with a single key. The object specification can be recursive, as the object can use other specifications to be the parameters.
- Instruct which jar to load from.
- Use an additional key to further specify which specification to use. This makes the keys more readable.
- Specify the specification factory

The heart of this component is the *ObjectFactory* class, again. It provides eight variations of the *createObject* method, each with varying ability to create an object. The most important parameter in each call is the *type*. This parameter can be any String as long as it maps to a valid specification, otherwise it must be the fully-qualified name of a valid class. The user can also pass a name and path to a JAR file from where the class will be loaded and created. The user also has the option to pass parameters to the constructor. But the most important aspect of this design is its ability to create complex objects from a specification, even if these complex objects require other complex objects as parameters.

This version also supports polymorphism. For example, assume we have a constructor XXClass(YYInterface), it's possible to create instances of XXClass using implementations/subclasses of YYInterface from configuration.

As an example, suppose there is a 2-param *Frac* class that takes an *int* and a *Bar* object. The *Bar* class takes two params: a *float* and a *StringBuffer*. The conventional instantiation would proceed as follows:

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

Once configured, this can be wrapped into a single call in this component:

```
Frac aFrac = (Frac) factory.createObject("frac", "default");
```

As such, the factory allows arbitrary configuration of an object, and all the user has to do is call for it. This will be especially helpful if the implementation of the

Frac is improved, sub-classed, packaged in a new jar, and constructor changed. The code will not have to change.

Generally speaking, simple objects are primitives, and complex objects are Object types. The exceptions are String, which is treated as a simple type, and arrays, which are treated as a special type of a complex object. This component supports arbitrary-dimension arrays in the configuration. It also supports passing null parameters programmatically and with specification.

The *ObjectFactory* is responsible for creating objects, but it obtains object definitions from the *SpecificationFactory*. This design comes with one implementation of such a factory – *ConfigManagerSpecificationFactory* – that is backed by the TC component ConfigManager. The specification factory is set when the *ObjectFactory* is created.

Although the *ObjectSpecification* used to transport the specifications from the specification factory to the object factory looks like it duplicates much of the reflection information, this design takes the approach of separating the handling of obtaining the specification from a source from the work of instantiating an object. This way, the *SpecificationFactory* implementations do not have to redo the logic of using reflection to instantiate an object, although at the cost of having to create a somewhat complex *ObjectSpecification*. The designer takes the approach that this trade-off is cost-effective.

Also the user should note that this component does not check for circular references. As such, the administrator should make sure the configuration is not circular.

1.1 Design Patterns

Factory: *ObjectFactory* is a factory of Objects.

Strategy: Used with the *SpecificationFactory* instance and with the instantiated objects.

1.2 Industry Standards

None

1.3 Required Algorithms

This section will show three 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.
- How the *ObjectFactory* uses an *ObjectSpecification* to recursively create an object instance.

The developer is encouraged to improve on these algorithms.

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());
```

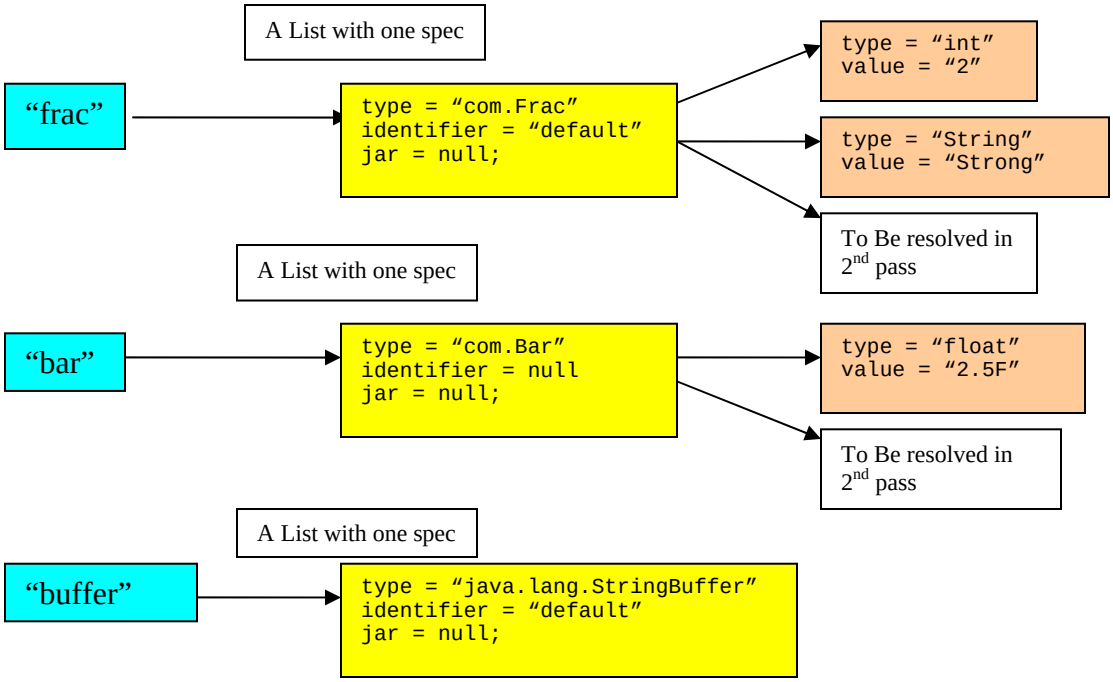
The configuration would look like the following (the file being stripped to the properties):

```
<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>
```

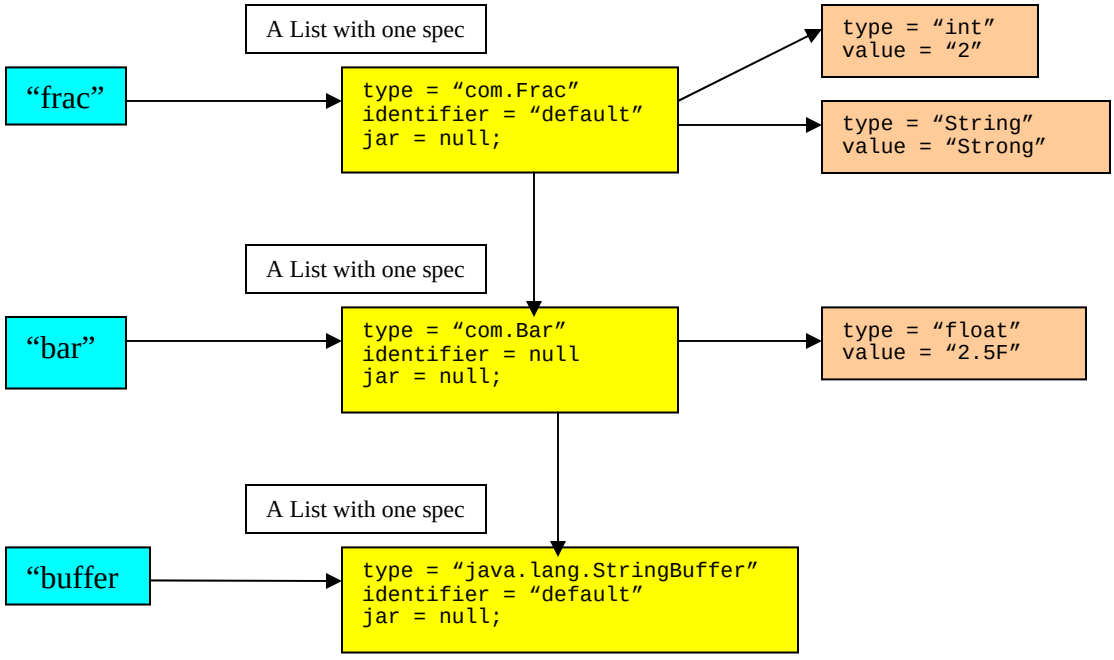
Following the algorithm, the first pass would result in 3 top-level entries in the map, with each entry being a List with one *ObjectSpecification* (Illustration 1), which contains the simple *ObjectSpecification* parameters:

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);
```

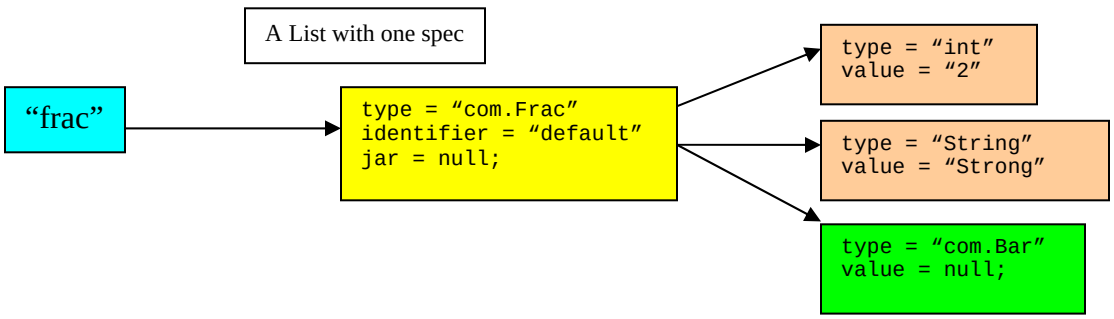
The null represents a null Bar object. The above configuration would change to the following:

```
<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 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



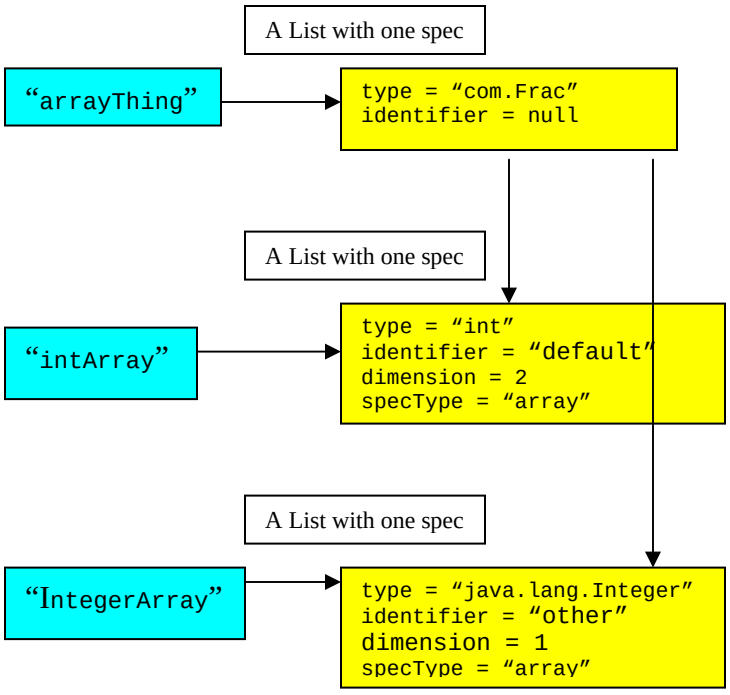
1.3.2 *Dealing with Arrays in configuration*

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.

1.3.5 *Instantiating object using an ObjectSpecification*

This section details how the *ObjectFactory* creates an object using an *ObjectSpecification* obtained from a specification factory.

The specification obtained from the specification factory might contain parameters that are also object specifications. As such, the entire specification represents the metadata for the recursive creation of an object. As noted before, there are three types of parameters: simple, complex, and complex array. Simple parameters will have a value and can be instantiated immediately, whereas all complex specifications will be made up of zero to more simple specifications, array specifications will contain arbitrary-dimension specifications, and nulls can be used immediately also. As such, the algorithm is a first-depth search for the simple parameters and nulls, and builds the instance from the bottom-up.

```
obtainInstance(spec)
    if type is simple or null {
        if type is primitive {
            wrap the value in its corresponding Object instance;
            return instance;
        } else if this is a 'String' or 'java.lang.String' {
           wrap the value as a String;
        } else if this is a null value {
           return null;
        }
    } else if spec is complex {
        if spec has param specs {
            obtainInstance(spec) for each param spec;
            call "instantiate a complex spec" using these obtained param
instances;
        } else {
            call "instantiate a complex spec" with no params;
        }
    } else if spec is array {
       obtainInstance(spec) for each param spec, replacing the specification
object with the instance;
```

```
        call "instantiate an array spec" with these params;
    }
    return instance;
}
```

This sample can be used to instantiate the complex object. Note that params can be null.

```
instantiate a complex spec (spec, params) {
  if spec contains a jar {
     create URLClassLoader with this jar as the sole URL;
     obtain a Class using this URLClassLoader with the type in spec;
  } else {
     obtain Class using the Class.forName(type);
  }

  obtain the Constructor from the Class object for these params;
    (simply use param[i].getClass(), or refer to spec for param[i] is null)
  using this Constructor, create an instance with these params;
  return the instance;
}
```

The array creation is very simple.

```
instantiate an array spec (spec, params) {
  create array instance using Array class with the dimensions of the params;
  set array values to the values in params;
  return the array instance;
}
```

The end result is a fully-instantiated object.

If the process fails at any point, throw *InvalidClassSpecificationException.*

Some notes on selecting the proper constructor. As can be seen in the algorithm for instantiating the complex object, the type of a null parameter can be obtained from the parameter's specification, which is available via *spec.getParameters()[i]*, with *i* being the index of this parameter.

Finally, since the object specification for the top-level object can be partially overridden programmatically, it is possible that the top-level list of parameters will be supplied, eliminating the need for using this algorithm and simply going directly to instantiating this complex object. In this case, either all *params* will be non-null and thus can be used to construct the Class[] or select the proper constructor, or the *paramTypes* will be available for that purpose. Either way there will always be no ambiguity about which constructor is to be used.

1.4 Component Class Overview

ObjectFactory:
The main class in this component. It is backed by a *SpecificationFactory*, which it queries for specifications for objects. Gives methods to create object where the user can specify two keys, the JAR file, and the parameters to use in the construction.

SpecificationFactory:
Interface to the factory that will supply specifications. Currently there is one implementation - *ConfigManagerSpecificationFactory*.

ObjectSpecification:
The object that contains the specification, or the metadata, that the *ObjectFactory* will use to instantiate objects.

ConfigManagerSpecificationFactory:
Concrete implementation of the *SpecificationFactory* backed by the *ConfigManager* as the source of the configuration. All specifications are loaded on startup.

1.5 Component Exception Definitions

This design creates seven custom exceptions in two hierarchies for ObjectFactory and SpecificationFactory

ObjectFactoryException:
Common exception for the ObjectFactory.

ObjectCreationException:
Common exception for exceptions that deal with the life cycle of creating an object.

InvalidClassSpecificationException:
Thrown by the *ObjectFactory createObject* methods if the specification is not valid and can't be used to create an object.

SpecificationFactoryException:
Common exception for the SpecificationFactory.

UnknownReferenceException:
The reference passed by the factory, which refers to the type and/or identifier, refers to a mapping that does not exist in the specification factory, or the specification tree for this reference contains mapping that does not exist. For example, the type and identifier might map to a valid specification, but one of its parameters might not. Thrown by the *getObjectSpecification* method in the *SpecificationFactory*.

SpecificationConfigurationException:
Throws by the *ConfigManagerSpecificationFactory* constructor if it cannot properly use the *ConfigManager*, i.e. the namespace is not recognized.

IllegalReferenceException:
Throws by the *ConfigManagerSpecificationFactory* constructor if it cannot properly match specifications to each other, or the properties are malformed.

1.6 Thread Safety

This component is generally thread-safe, as no regular action by one thread will have an effect on the action of another, as in most cases, the objects are immutable. Even in the case of the *initStrategy* member of the *ObjectFactory* where a thread could change this value while another thread is creating an object using this value, there would be no adverse effect, because the reading of this value is atomic.

One mention should be made about the arrays used in the *ObjectSpecification* and *ObjectFactory* classes, although this does not strictly fall into the category of thread-safety. It is possible that one thread could get a hold of these and make changes to their contents while another is using them, potentially causing inconsistencies. This scenario, however, is not expected to occur even by mistake, as generally, the object parameters to pass to the *ObjectFactory* are not expected to change after they are created, and it is not expected that a thread will intentionally change the *params* in the *ObjectSpecification*. If an application using this component does intend to change them in separate threads, then it must take these issues into consideration.

2. Environment Requirements

2.1 Environment

- At minimum, Java 1.4 is required for compilation and executing test cases.

2.2 TopCoder Software Components

- Config Manager 2.1.4
    - Used to specify object definitions in the *ConfigManagerSpecificationFactory*.
- Base Exception 1.0
    - Provides common base exception for all TC components, and is a standard for all TC components.

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
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)

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 Configuration Manager and Base Exception components must be added, as well as any extra jars that will be used in the course of creating objects.

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 *Convenience method demo*

This section details the use of the four convenience methods. A representative sample of various possible uses is shown.

```
        // instantiate factory with specification factory
        ObjectFactory factory =
            new ObjectFactory(new
ConfigManagerSpecificationFactory("valid_config"), ObjectFactory.BOTH);

        // obtain the configured default frac
        TestClass1 aClass = (TestClass1) factory.createObject("frac",
"default");

        // obtain the TestClass2, without identifier.
        TestClass2 aFrac = (TestClass2) factory.createObject("bar");

        // change initialization strategy
        factory.setInitStrategy(ObjectFactory.REFLECTION_ONLY);

        // obtain com.test.TestComplex object in specified jar file. Will
use reflection only.
        Object testComplex =
factory.createObject("com.topcoder.util.objectfactory.testclasses.TestCla
ss2");
```

4.3.3 *Main method demo*

This section details the use of the four main 6-param methods. A representative sample of various possible uses is shown.

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

        // obtain the configured bar, without using the identifier, and
rest as defaults
        TestClass2 bar =
            (TestClass2) factory.createObject("bar", null, (ClassLoader)
null, null, null, ObjectFactory.BOTH);

        // obtain TestComplex object, but use this jar and parameters
instead,
        Object[] params = {new Integer(12), "abc"};
        URL url = new
URL(TestHelper.getURLString("test_files/test.jar"));
        Class[] paramTypes = {int.class, String.class};
        Object complex =
            factory.createObject("com.test.TestComplex", null, url,
params, paramTypes, ObjectFactory.BOTH);

        // obtain another TestClass1 object with the same params, but
just use reflection
        TestClass1 bar2 =
            (TestClass1) factory.createObject(TestClass1.class, null,
(ClassLoader) null, params, paramTypes,
```

```
                    ObjectFactory.REFLECTION_ONLY);

            // obtain Collection array, but just use specification
            Collection[] bar3 =
                (Collection[]) factory.createObject(int.class, "collection",
(ClassLoader) null, params,
                    paramTypes, ObjectFactory.SPECIFICATION_ONLY);

            // obtain TestClass1 object using relfection from a specified
ClassLoader, using
            // params but no paramTypes since no nulls used.
            params = new Double[] {new Double(12.00)};

            ClassLoader loader = ClassLoader.getSystemClassLoader();

            TestClass1 bar4 =
                (TestClass1) factory.createObject(TestClass1.class, null,
(ClassLoader) null, params, null,
                    ObjectFactory.REFLECTION_ONLY);
```

5. Future Enhancements

- Provide other SpecificationFactory implementations to deal with different sources.
- Provide additional initialization hooks. This would help if an object needs post-construction initialization.