Object Factory 2.2 Component Specification

1. 1. Design

All changes performed when synchronizing documentation with the version 2.1.2 of the source code of this component and fixed errors in the CS are marked with **purple**.

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

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

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*.

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.

Changes in the version 2.2:

- • 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. 1.1 Design Patterns

Strategy pattern – ObjectFactory uses pluggable SpecificationFactory implementation instance.

1. 1.2 Industry Standards

None

1. 1.3 Required Algorithms

This section will show one algorithm:

1. • How the *ObjectFactory* uses an *ObjectSpecification* to recursively create an object instance.

1. 1.3.1 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. 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.
Changes in 2.2:
- Made some private methods static.
- Specified generic parameters for all generic types in the code.

SpecificationFactory:
Interface to the factory that will supply specifications. This component doesn't provide implementations of this interface.

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

1. 1.5 Component Exception Definitions

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

ObjectFactoryException:

Common exception for the ObjectFactory.
Changes in 2.2:
- Extends BaseCriticalException instead of BaseException.
- Added new constructors to meet TopCoder standards.

ObjectCreationException:
Common exception for exceptions that deal with the life cycle of creating an object.
Changes in 2.2:
- Added new constructors to meet TopCoder standards.

InvalidClassSpecificationException:
Thrown by the *ObjectFactory createObject* methods if the specification is not valid and can't be used to create an object.
Changes in 2.2:
- Added new constructors to meet TopCoder standards.

SpecificationFactoryException:
Common exception for the SpecificationFactory.
Changes in 2.2:
- Extends BaseCriticalException instead of BaseException.
- Added new constructors to meet TopCoder standards.

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*.
Changes in 2.2:
- Added new constructors to meet TopCoder standards.

SpecificationConfigurationException:
Can be thrown by constructor of SpecificationFactory implementations if some configuration specific error occurred.
Changes in 2.2:
- Added new constructors to meet TopCoder standards.

IllegalReferenceException:
Can be thrown by constructor of SpecificationFactory implementations if cannot properly match specifications to each other, or the properties are malformed.
Changes in 2.2:
- Added new constructors to meet TopCoder standards.

1. 1.6 Thread Safety

This component is generally thread-safe when the used SpecificationFactory implementation is thread safe (taking into account that ObjectSpecification instances are used by this component in thread safe manner), 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 (this is true in the version 2.2 only since previously initStrategy field had no "volatile" modifier).

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.

Except fixing an atomic ObjectFactory#initStrategy assess issue, thread safety of the component was not changed in the version 2.2.

1. 2. Environment Requirements

2. 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

1. 2.2 TopCoder Software Components

- **Base Exception 2.0**
  - 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.

1. 2.3 Third Party Components

None

1. 3. Installation and Configuration

2. 3.1 Package Name

com.topcoder.util.objectfactory
com.topcoder.util.objectfactory.impl

1. 3.2 Configuration Parameters

None

1. 3.3 Dependencies Configuration

None

1. 4. Usage Notes

2. 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.

1. 4.2 Required steps to use the component

No special steps are required to use this component.

1. 4.3 Demo

1. 4.3.1 Setup

For the purposes of the demo, we will assume the use of the ConfigManagerSpecificationFactory found in the Object Factory Config Manager Plugin component, but any plugin could be used, which would entail skipping this sub-section in lieu of the selected plugin’s configuration

We can start with the specification 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>
```

1. 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.TestClass2");
```

1. 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<Object>[] bar3 = (Collection<Object>[]) factory.createObject(Collection.class,
"collection", (ClassLoader) null, params, paramTypes, ObjectFactory.SPECIFICATION_ONLY);

// obtain TestClass1 object using reflection 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, loader, params, null,
ObjectFactory.REFLECTION_ONLY);
```

1. 5. Future Enhancements

1. • Provide new SpecificationFactory implementations to deal with different sources.
2. • Provide additional initialization hooks. This would help if an object needs post-construction initialization.
3. • Add ability to use the factory in reflection-only mode, without the need to specify a specification factory. This can be accomplished by adding a parameter-less constructor which will set the mode to reflection-only, and this should also disable the ability to reset the mode.