Standard Class Data Loader 1.0 Component Specification

1. Design

With many modern programming languages, each comes with its own set of 'standard' classes and namespaces of typical classes or functions that a user may wish to use. Examples of this are the `java.*` packages, the `System` namespace in C#, and the standard library in C/C++ code - `stdlib` or `std::` respectively.

When developing with these languages, the process can be improved if a user has access to information about these standard features - this may be a public API, or in the case of a UML application, simply a list of namespaces and their contents which can be used within a model.

This component is designed for the latter use: to read in a tree of namespaces and their contents - Classes, Interfaces, Primitives, DataTypes and Enumerations. This is done by loading the data from an XML file with a simplified format, which allows any user to add or remove namespaces or classifiers as they require. This XML loading backs a generic loading strategy, which allows different persistence methods to be plugged in, as well attaching different loaders to a variety of languages.

1.1 Design Patterns

Strategy: The ClassDataLoaderStrategy class uses the strategy pattern in providing interchangeable loading algorithms to use to load namespace information, as well as allowing different ones to be used for each language.

Factory Method: The XMLLanguageClassDataLoader class provides simple loadXXX methods that build UML model elements from XML data - they have been given protected access, allowing subclasses to override the methods as different ways of parsing the elements is desired.

1.2 Industry Standards

XML is used to load the UML namespace information in from a persistent file.

1.3 Required Algorithms

The complex algorithms within this component all relate to the way XML Element nodes are translated into their ModelElement counterparts. There are three different ways this is done currently - each is detailed in the Sample Load Algorithms Sequence Diagram. They are:

```
1 - Namespace loading:
Create a new PackageImpl, setting its name to the “name” attribute.
For each child node:
      Find the tag name of the child node
      Use the appropriate loadXXX method to convert to ModelElement
      Add this by calling PackageImpl#addOwnedElement(...)

2 - Class (or Interface) loading:
Create a new ClassImpl, setting its name to the “name” attribute.
Load the child nodes (namespace / class / primitive, etc...) as above

For each child “Method” node:
      Create a new MethodImpl, setting its name to the “name” attribute.
      Add the method to the class using ClassImpl#addFeature(...)


3 - Primitive (or DataType or Enumeration) loading:
Create a new PrimitiveImpl, setting its name to the “name” attribute
```

1.4 Component Class Overview

StandardClassDataLoader << interface >>:

Interface defining the single operation required for standard data class loaders. Each implementation should be threadsafe, and calling a getNamespaces() method should have no side effects.

ClassDataLoaderStrategy:

Implementation of the Standard Class Data loader which allows each language to be attached to its own pluggable loading algorithm. This is achieved by maintaining a mapping from known languages to their respective LanguageClassDataLoaders. When the getNamespaces method is called, the language is then used as a key into the map, and the relevant language loader algorithm is used to retrieve the namespaces. The mappings are loaded on construction. This class is threadsafe - after construction, it only performs retrieval methods on its loaders, while ensuring each loader is called by one thread at a time.

LanguageClassDataLoader << interface >>:

Interface defining the single operation required for data class loaders of a single language. This provides a single method that retrieves the namespace information for the language the loader is responsible for. Implementations do not have to be threadsafe - however, they must be able to be constructed using the Object factory if they are to be used by the ClassLoaderStrategy.

XMLLanguageClassDataLoader:

Implementation of the Language Class Data Loader that loads the data from an XML file, whose format is specified by the provided xml schema (*schema.xml*). The file is created on construction and parsed into a DOM document. Then, a variety of methods are provided to help convert XML Elements into the various ModelElement implementations that they represent. This class handles its own thread safety to the greatest extent it can - after construction, its state never changes. The only method that can be called post construction is a getter which only returns a copy of the single member.

1.5 Component Exception Definitions

ClassDataLoaderConfigurationException:

Exception thrown whenever there is a problem initializing a data class loader or any classes it relies on. This can be thrown from the constructors of implementations of StandardDataClassLoader.

UnknownLanguageException:

Exception thrown whenever a language is passed to a standard loader, which is not recognized by the loader. This is raised by the getNamespaces method of StandardDataClassLoader, and any of its implementations, for whenever the language required is not known.

1.6 Thread Safety

This component is thread safe - the loader provides a lock on the namespaces list when it is first being built, and from then it is not modified. To protect against possible later implementations that are not thread safe, the Class data loader strategy also applies a synchronize lock on any language loader before obtaining namespaces from it.

2. Environment Requirements

2.1 Environment

- Java 1.5 for compilation and development

2.2 TopCoder Software Components

- Base Exception 1.0 for the base class of the component's custom exceptions.
- Configuration Manager 2.1.5 for loading in language configuration list.
- Object Factory 2.0.1 for generation of LanguageClassDataLoader instances.
- UML Model - Core 1.0
- UML Model - Core Classifiers 1.0
- UML Model - Model Management 1.0 all for required data structures.

2.3 Third Party Components

- The org.w3c.dom package is used to handle the XML using DOM techniques.

3. Installation and Configuration

3.1 Package Name

com.topcoder.uml.standardclassloader
com.topcoder.uml.standardclassloader.implementations

3.2 Configuration Parameters

Parameter	Description	Values
objectFactoryNamespace *[required]*	Namespace to load object factory information from	Valid namespace for an Object factory.
languages [optional*]*	List of languages names supported by the strategy.	0 or more non-empty strings.
language/key *[required]*	Key for Object factory creation of Class data loader for *language*.	Valid object factory key.

Provided is a file, *sampleConfig.xml*, showing a possible configuration file for use when a Java XML loader is desired. Along with this is an example file of class data - *sampleJavaFile.xml* as well as the schema that the XMLLanguageClassDataLoader is expecting: *schema.xml*.

3.3 Dependencies Configuration

The configuration manager component may require initialization to know where to read configuration from - please consult its component specification for more details.

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

See below for example code on how this component is used.

4.3 Demo

4.3.1 It shows the functionality of the ClassDataLoaderStrategy.

```
// construction using the sample configuration:
StandardClassDataLoader loader = null;
try {
    // constructs a new loader, which uses the object factory
    // to initialize the "Java" loader using the factory key "javaLoader"
    loader = new ClassDataLoaderStrategy("strategyClassDataLoader");
} catch (ClassDataLoaderConfigurationException cdlce) {
    System.err.println("shouldn't happen...");
}

// should throw an exception if using an unrecognized language:
try {
    loader.getNamespaces("VB.NET");
} catch (UnknownLanguageException e) {
    // will print "The language [VB.NET] is unknown."
    System.err.println(e.getMessage());
}

// make sure we have the java loader:
ClassDataLoaderStrategy strategy = (ClassDataLoaderStrategy) loader;
assertTrue("The language [Java] should be loaded.",
strategy.getLanguages().contains("Java"));

// check some values in the list for java:
List<Namespace> nslist = loader.getNamespaces("Java");

// one top-level namespace and four primitives (using sample config)
assertTrue("Expected the size of the list is five.", nslist.size() == 5);
assertTrue("Expected the name of the first namespace is [java].",
nslist.get(0).getName().equals("java"));
assertTrue("Expected the name of the second namespace is [boolean].",
nslist.get(1).getName().equals("boolean"));
assertTrue("Expected the name of the third namespace is [char].",
nslist.get(2).getName().equals("char"));
assertTrue("Expected the name of the fourth namespace is [int].",
nslist.get(3).getName().equals("int"));
assertTrue("Expected the name of the fifth namespace is Primitive instance.",
nslist.get(4) instanceof Primitive);
```

4.3.2 It shows the functionality of the XMLLanguageClassDataLoader.

```
// create a XMLLanguageClassDataLoader instance
XMLLanguageClassDataLoader failed = new
XMLLanguageClassDataLoader("noSuchFile.txt");
try {
    failed.getNamespaces();
} catch (ClassDataLoaderConfigurationException cdlce) {
    System.err.println("This should fail, need correct file format.");
}

// correctly configure an XML loader:
XMLLanguageClassDataLoader xmlLoader = new
XMLLanguageClassDataLoader("test_files" + File.separator
    + "sampleJavaFile.xml");

// check that it does the same as before:
List<Namespace> nslist = xmlLoader.getNamespaces();

assertTrue("Expected the size of the list is five.", nslist.size() == 5);
assertTrue("Expected the name of the first namespace is [java].",
nslist.get(0).getName().equals("java"));
assertTrue("Expected the name of the second namespace is [boolean].",
nslist.get(1).getName().equals("boolean"));
assertTrue("Expected the name of the third namespace is [char].",
nslist.get(2).getName().equals("char"));
assertTrue("Expected the name of the fourth namespace is [int].",
 nslist.get(3).getName().equals("int"));
assertTrue("Expected the name of the fifth namespace is Primitive instance.",
nslist.get(4) instanceof Primitive);
```

5. Future Enhancements

One place to start enhancements could be to allow more information to be read in from the XML file - currently, beyond simple ‘name’ values, the only extension to this are the nested namespaces and method names for classes or interfaces. More data can be loaded if required, such as parameter lists for methods, or values within enumerations.

Another enhancement if speed is an issue would be to allow lazy instantiation of language loaders - that way, the strategy could know about many languages but would only have to go through the process of loading the data for each that is actually used.