UML Tool JAR Importer 1.0 Component Specification

1. Design

The UML Tool currently has no way of importing data into the tool an external source. This component will allow for the import of Java class data into the tool, into an existing diagram. This will enable users to write test Java code and just import it into the diagram. This component will import the data directly into the UML Tool project model.

1.1 Design Patterns

Template Method Pattern

The DefaultJarImporter extends JarImporter and implements the template methods retrievePackagesAndClassesFromJar, extractExternalClasses, importPackageAndClassesToModel and importToDiagram, it is a Template Method Pattern.

1.2 Industry Standards

JAR

1.3 Required Algorithms

1.3.1 Retrieve packages and classes from JAR file.

This algorithm is for method DefaultJarImporter#retrievePackagesAndClassesFromJars. It uses URLClassLoader to load the classes and packages.

```
// Create an URLClassLoader by the given URL and classpathLoader which is
// initialized in the constructor.
URLClassLoader loader = createURLClassLoader(jarURLs, classpathLoader);

// for each jar
for (URL jarURL : jarURLs) {
// Create JarFile instance
JarFile jar = createJarFile(jarURL);

// for each jar entry
for (Enumeration<JarEntry> e = jar.entries(); e.hasMoreElements();) {
// Get the name of class, the format will like : 'com/topcoder/pack/MyEnum.class'
String name = e.nextElement().toString();

// Filter the .MF file and directory etc.
if (name.endsWith(CLASS)) {
// it is a class entry, get the class binary name
name = name.substring(0, name.length() - 6).replace('/', '.');
Class<?> clazz;
try {
clazz = loader.loadClass(name);
```

```
} catch (Throwable cnfe) {
// When the class is invalid, java.lang.ClassFormatError may be thrown.
throw new JarImporterException("Error when loading class with name'" + name
+ "' for " + cnfe.getMessage(), cnfe);
}

addClassAndPackage(getClasses(), clazz, name);
}
}
}
```

1.3.2 Extract external classes

This algorithm is for method DefaultJarImporter#extractExternalClasses.

```
// For each class in classes, get the external classes
for (Class<?> clazz : getClasses().values()) {
// Add all the interfaces implemented by this class and its package to mappings.
for (Class<?> inter : clazz.getInterfaces()) {
addClassAndPackage(getExternalClasses(), inter, inter.getName());
}

// Add the super class and package to mappings.
Class<?> superclass = clazz.getSuperclass();
if (superclass != null) {
addClassAndPackage(getExternalClasses(), superclass, superclass.getName());
}

// Add all the fields type class and package to mappings.
for (Field field : clazz.getDeclaredFields()) {
addClassAndPackage(getExternalClasses(), field.getType(), field.getType().getName());
}

// Add all the methods, return type class and package to mappings.
for (Method m : clazz.getDeclaredMethods()) {
// Add each method parameter type and package to mappings.
for (Class<?> paramClass : m.getParameterTypes()) {
addClassAndPackage(getExternalClasses(), paramClass, paramClass.getName());
}

// Add the return type class and package to mappings.
addClassAndPackage(getExternalClasses(), m.getReturnType(), m.getReturnType()
.getName());
}
        }
```

```
private void addClassAndPackage(Map<String, Class<?>> classes, Class<?> clazz, String
clazzName) {

// http://forums.topcoder.com/?module=Thread&threadID=620923&start=0
// the 'void' type should be removed.
if (VOID.equals(clazzName)) {
return;
}

if (!this.getClasses().containsKey(clazzName)) {
// Add the classes containing 'int', 'double' etc to external classes mapping.
classes.put(clazzName, clazz);

java.lang.Package pkg = clazz.getPackage();

if (pkg == null) {
getPackages().put("", null);
} else {
if (!getPackages().containsKey(pkg.getName())) {
getPackages().put(pkg.getName(), pkg);
}
}
}
                }
```

1.3.3 Import entities into model

This algorithm is for method DefaultJarImporter#importPackagesAndClassesToModel.

1. Add the packages to the model
2. Add the external classes to the model
3. Add the classes in the jar to the model
4. Add features (attributes and methods) and relationships (association, dependency, abstraction, generalization) of classes in jar to model, in this step, the relationships will be also added to the relationships list which may to be added to diagram.

```
Model model = this.getModelManager().getModel();

// ////////////////////////////////////////////////////////////////////////////
// Part1 Add packages to model
// For each package, add it to model, the 'java.*' package should not filter since
// these packages can be in model.
for (String packageName : getPackages().keySet()) {
```

```
        // The package name can not be whitespaces.
        // http://forums.topcoder.com/?module=Thread&threadID=620926&start=0
        if (packageName.trim().length() > 0) {
        Package p = new PackageImpl();
        p.setName(packageName);
        p.setNamespace(model);
        model.addOwnedElement(p);
        getModelPackages().put(packageName, p);
        } else {
        // default package, add model as the package
        getModelPackages().put(packageName.trim(), model);
        }
        }

        // ////////////////////////////////////////////////////////////////////////////
        // Part2 Add external classes/interfaces/datatypes to model (getModelClasses()).
        for (Class<?> clazz : getExternalClasses().values()) {
        importClassesToModule(clazz, model);
        }

        // ////////////////////////////////////////////////////////////////////////////
        // Part3 Add classes/interfaces from the jar to model (getModelClasses())
        for (Class<?> clazz : getClasses().values()) {
        importClassesToModule(clazz, model);
        }

        // ////////////////////////////////////////////////////////////////////////////
        // Part4 Add features (attributes and methods) and relationships (association,
        // dependency, abstraction, generalization) of classes to model
        importClassAttributesToModule(model);

private void importClassesToModule(Class<?> clazz, Model model) throws JarImporterException {
// Create corresponding Classifier instance for the class.
// The class can be interface or enum or primitive or class.
// Note : http://forums.topcoder.com/?module=Thread&threadID=620926&start=0
// array type should be ignored now and it should be done during final fix.
Classifier classifier = null;
if (clazz.isInterface()) {
classifier = new InterfaceImpl();
} else if (clazz.isEnum()) {
classifier = new EnumerationImpl();
} else if (clazz.isPrimitive()) {
```

```
// for inner class, it never happens.
classifier = new PrimitiveImpl();
} else if (clazz.isArray()) {
// ignored now.
throw new JarImporterException("array is not supported.");
} else {
classifier = new ClassImpl();
}

// set the namespace of it, if package is null, set model as its namespace.
java.lang.Package javaPackage = clazz.getPackage();
if (javaPackage == null) {
classifier.setNamespace(model);

// model will be the owned element for classifier.
model.addOwnedElement(classifier);
} else {
Package p = (Package) getModelPackages().get(javaPackage.getName());
classifier.setNamespace(p);

// add it to the namespace
p.addOwnedElement(classifier);
}

classifier.setName(clazz.getSimpleName());

// set 'final', 'static', and 'abstract' modifiers properties to classifier
int mod = clazz.getModifiers();
classifier.setAbstract(Modifier.isAbstract(mod));
classifier.setLeaf(Modifier.isFinal(mod));
classifier.setRoot(Modifier.isStatic(mod));

// Set the visibility of classifier. Only private, protected and public are supported.
setVisibility(classifier, mod);

// add all the classes and external classes to model classes.
// Note: getModelClasses() will contain these class : java.*
getModelClasses().put(clazz.getName(), classifier);
}
private void importClassAttributesToModule(Model model) {
for (Class<?> cls : getClasses().values()) {
Classifier classifier = getModelClasses().get(cls.getName());
```

```
java.lang.Package pkg = cls.getPackage();
String packageName = pkg == null ? "" : pkg.getName();

// if package name is empty string, model will be retrieved from getModelPackages().
Package p = getModelPackages().get(packageName);

List<Relationship> relationships = new ArrayList<Relationship>();

// add the relationship to relations list.
addRelationWithInterfaces(model, cls, classifier, packageName, p, relationships);
addRelationWithSuperClass(model, cls, classifier, packageName, p, relationships);
addRelationWithFields(model, cls, classifier, packageName, p, relationships);
addRelationWithMethods(model, cls, classifier, packageName, p, relationships);
addRelationWithInnerClass(model, cls, classifier, packageName, p, relationships);

this.getClassesToRelationships().put(classifier, relationships);
}
}

private void addRelationWithInterfaces(Model model, Class<?> cls, Classifier classifier,
String packageName, Package p, List<Relationship> relationships) {
for (Class<?> inter : cls.getInterfaces()) {
// Create an Abstraction
Abstraction abstraction = new AbstractionImpl();

// add supplier of this abstraction
abstraction.addSupplier(getModelClasses().get(inter.getName()));

// add client of this abstraction
abstraction.addClient(classifier);

// set the namespace to.
setNamespace(inter, model, abstraction, packageName, p);

classifier.addClientDependency(abstraction);

getRelationships().add(abstraction);

// add all the relationship with cls to list.
relationships.add(abstraction);
}
```

```
}

private void setNamespace(Class<?> clazz, Model model, ModelElement ele, String packageName,
Package p) {
if (clazz.getPackage() == null) {
ele.setNamespace(model);
} else {
ele.setNamespace(packageName.equals(clazz.getPackage().getName()) ? p : model);
}
}

private void setVisibility(ModelElement ele, int mod) {
if (Modifier.isPrivate(mod)) {
ele.setVisibility(VisibilityKind.PRIVATE);
} else if (Modifier.isProtected(mod)) {
ele.setVisibility(VisibilityKind.PROTECTED);
} else if (Modifier.isPublic(mod)) {
ele.setVisibility(VisibilityKind.PUBLIC);
}
}

private void addRelationWithSuperClass(Model model, Class<?> cls, Classifier classifier,
String packageName, Package p, List<Relationship> relationships) {
Class<?> superclass = cls.getSuperclass();
if (superclass != null) {
// Create a Generalization
Generalization generalization = new GeneralizationImpl();
// set parent of the Generalization
generalization.setParent(getModelClasses().get(superclass.getName()));
// set child of the Generalization
generalization.setChild(classifier);

// set the namespace to.
setNamespace(superclass, model, generalization, packageName, p);

classifier.addGeneralization(generalization);
getRelationships().add(generalization);

// also add it to the classesToRelationships map
// this.getClassesToRelationships().get(classifier).add(generalization);
relationships.add(generalization);
}
```

```
}

private void addRelationWithFields(Model model, Class<?> cls, Classifier classifier,
String packageName, Package p, List<Relationship> relationships) {
for (Field field : cls.getDeclaredFields()) {
// Here should be the type of field, not using getDeclaringClass.
Class<?> fieldClass = field.getType();

// add the field as the attribute in the classifer
Attribute attribute = new AttributeImpl();
attribute.setType(getModelClasses().get(fieldClass.getName()));
attribute.setOwner(classifier);
attribute.setName(field.getName());

// If the field is final, set to frozen.
// Note : the static attribute should be done during final fix.
int fieldMod = field.getModifiers();
if (Modifier.isFinal(fieldMod)) {
attribute.setChangeability(ChangeableKind.FROZEN);
}

// Set the visibility of classifier. Only private, protected and public are supported.
setVisibility(attribute, fieldMod);

classifier.addFeature(attribute);

// the relationship to primitive and system classes should be ignored.
if (!fieldClass.isPrimitive() && !fieldClass.getName().startsWith(JAVAPREF)) {
// create AssociationEnd instances
AssociationEnd end1 = new AssociationEndImpl();
end1.setParticipant(classifier);
AssociationEnd end2 = new AssociationEndImpl();
end2.setParticipant(getModelClasses().get(fieldClass.getName()));
end2.setName(field.getName());

// also set visibility to end2.
setVisibility(end2, fieldMod);

// create Association instances
Association association = new AssociationImpl();
association.addConnection(end1);
association.addConnection(end2);
```

```
end1.setAssociation(association);
end2.setAssociation(association);

// set the namespace to.
setNamespace(fieldClass, model, association, packageName, p);

getRelationships().add(association);

// also add it to the classesToRelationships map
// this.getClassesToRelationships().get(classifier).add(association);
relationships.add(association);
}
}

}

private void addRelationWithMethods(Model model, Class<?> cls, Classifier classifier,
String packageName, Package p, List<Relationship> relationships) {
for (Method m : cls.getDeclaredMethods()) {
Operation operation = new OperationImpl();
operation.setName(m.getName());
operation.setOwner(classifier);

// set final, static, and abstract modifiers properties
int mod = m.getModifiers();

operation.setAbstract(Modifier.isAbstract(mod));
operation.setLeaf(Modifier.isFinal(mod));
operation.setRoot(Modifier.isStatic(mod));

// Set the visibility of operation. Only private, protected and public are supported.
setVisibility(operation, mod);

// add parameters to this operation
for (Class<?> paramClass : m.getParameterTypes()) {
Parameter parameter = new ParameterImpl();
// Note that we can't get parameter name from compiled
// class by reflection. So we use the paramClass's name as
// the name.
String name = paramClass.getSimpleName();
// if the class is only one letter.
if (name.trim().length() == 1) {
```

```
name = name.toLowerCase();
} else {
name = name.substring(0, 1).toLowerCase() + name.substring(1);
}

parameter.setName(name);
parameter.setType(getModelClasses().get(paramClass.getName()));

operation.addParameter(parameter);
}

// also add Return as the parameter with the name "Return" to operation.
Class<?> retType = m.getReturnType();
Parameter ret = new ParameterImpl();
ret.setName(RETURN);
ret.setType(getModelClasses().get(retType.getName()));
operation.addParameter(ret);

// Iterate the parameter of the method
for (Class<?> paramClass : m.getParameterTypes()) {
createDependency(model, paramClass, classifier, packageName, p, relationships);
}

// set the dependency for return type.
createDependency(model, retType, classifier, packageName, p, relationships);
}
}

private void createDependency(Model model, Class<?> cls, Classifier classifier,
String packageName, Package p, List<Relationship> relationships) {
if (!cls.isPrimitive() && !cls.getName().startsWith(JAVAPREF)) {
Dependency dependency = new DependencyImpl();
Classifier c = getModelClasses().get(cls.getName());
if (c != null) {
// add supplier of this dependency
dependency.addSupplier(c);
}

// add client of this dependency
dependency.addClient(classifier);

// set the namespace to.
```

```
setNamespace(cls, model, dependency, packageName, p);

classifier.addClientDependency(dependency);
getRelationships().add(dependency);

// also add it to the classesToRelationships map
// getClassesToRelationships().get(classifier).add(dependency);
relationships.add(dependency);
}
}

private void addRelationWithInnerClass(Model model, Class<?> cls, Classifier classifier,
String packageName, Package p, List<Relationship> relationships) {

// getClasses method can not get the inner class.
// getDeclaredClasses is the correct method.
for (Class<?> inner : cls.getDeclaredClasses()) {
Dependency dependency = new DependencyImpl();

Classifier c = getModelClasses().get(inner.getName());
if (c != null) {
// add supplier of this dependency
dependency.addSupplier(c);
}

// add client of this abstraction
dependency.addClient(classifier);

Stereotype stereotype = new StereotypeImpl();
stereotype.setName(INNER);

// Add it to model first.
if (!model.containsOwnedElement(stereotype)) {
model.addOwnedElement(stereotype);
}

dependency.addStereotype(stereotype);

// set the namespace to.
setNamespace(inner, model, dependency, packageName, p);

classifier.addClientDependency(dependency);
getRelationships().add(dependency);
```

```
// also add it to the classesToRelationships map
// getClassesToRelationships().get(classifier).add(dependency);
relationships.add(dependency);
}
}
```

1.3.4 Add the imported classes into the specific diagram

This algorithm is for method DefaultJarImporter#importToDiagram.
It is straightforward that just iterate the modelClasses and relationships, for each classifier or relationship, add one GraphNode/GraphEdge to the specific diagram. If the specific diagram doesn't exist in the UMLModelManager, just throw JarImporterException.

```
// add the relationships to the diagram first
// define the Relationship to GraphEdge map.
Map<Relationship, GraphEdge> edges = new HashMap<Relationship, GraphEdge>();
for (Relationship ship : getRelationships()) {
GraphEdge edge = new GraphEdge();
Uml1SemanticModelBridge brige = new Uml1SemanticModelBridge();
brige.setElement(ship);
edge.setSemanticModel(brige);
edge.setPosition(new Point());

diagram.addContained(edge);
edges.put(ship, edge);
}

// add the classifiers to the diagram
for (Classifier classifier : getModelClasses().values()) {

// the primitive class and system classes should not be imported.
String name = classifier.getNamespace().getName();
if (name == null || name.startsWith(JAVAPREF)) {
continue;
}

// Create the node for each classifier.
GraphNode node = new GraphNode();
node.setPosition(new Point());
Uml1SemanticModelBridge brige = new Uml1SemanticModelBridge();
brige.setElement(classifier);
node.setSemanticModel(brige);
```

```
// get related Relationships for this classifier
List<Relationship> rs = getClassesToRelationships().get(classifier);

// the relationships may be null. For example, A connect to B, the relationship
// is owned by B, and when passing A, the return relationship is null.
if (rs != null) {
for (Relationship r : rs) {
GraphConnector connector = new GraphConnector();
GraphEdge edge = edges.get(r);
connector.addGraphEdge(edge);
node.addAnchorage(connector);
edge.addAnchor(connector);
}
}

// add it to diagram
diagram.addContained(node);
}
```

1.4 Component Class Overview

JarImporter:

This abstract class defines the APIs to import JARs to TC UML model and diagram. It defines three steps to achieve this purpose, first, pull all the classes and packages from the JARs; second, import the packages, classes, related external classes, relationships to the model; third, import all the entities pulled from JARs into a specific diagram. The implementation of them is left to the subclasses. This class also defines many variables to store the state of the importing.
Thread Safety: This class and the subclasses are not thread-safe because it has state when importing.

DefaultJarImporter:

This class is the default implementation of the JarImporter abstract class. It uses URLClassLoader and JarFile to pull the packages and classes from the JARs, it uses reflection to get the external classes, related relationships, attributes, operations, etc. And then import them into TC UML Model. At last, it just put all the classes and relationships into the specific diagram, the positions in the diagram are not specified. It also allows the user specify an extra classpath so that the target JARs can be imported successfully. Otherwise, some external classes may be unable to be loaded by reflection.
Thread Safety: This class is not thread-safe as the base class is not thread-safe.

1.5 Component Exception Definitions

IllegalArgumentException [System]:

This exception is thrown if any argument is invalid, for example: parameter is null or string is empty, etc.

JarImporterException [Custom]:

This is a custom exception if any errors occurred when importing the jar to the TC UML model and diagram.

1.6 Thread Safety

This component is not thread-safe because the JarImporter has inner state when importing the JARs.

To make this component completely thread safe synchronization will have to be heavily used in all blocks that mutate internal states, or just creates JarImporter instance for every caller thread.

2. Environment Requirements

2.1 Environment

? Development language: Java 5.0
? Compile target: Java 5.0

2.2 TopCoder Software Components

Base Exception 2.0: JarImporterException extends BaseCriticalException which is in Base Exception 2.0 component.
Diagram Interchange 1.0.1: It defines Diagram, GraphNode, etc classes.
UML Model - Core Dependencies 1.0: It defines Abstraction, Dependency, etc.
UML Model - Core Relationships 1.0: It defines Relationship, Association, etc.
UML Model - Core 1.0: It defines Attribute, Parameter, Classifier, etc.
UML Model - Model Management 1.0: It defines Model, Package, etc.
UML Model - Core Classifiers 1.0: It defines Class, Interface, etc.
UML Model - UML Model Manager 1.0: UMLModelManager is in that component.
UML Model – Data types 1.0: It defines data type etc.
UML Model –Core Extension mechanisms 1.0: It defines Stereotype class etc.

2.3 Third Party Components

None.

3. Installation and Configuration

3.1 Package Name

com.topcoder.uml.importer.jarimporter
com.topcoder.uml.importer.jarimporter.impl

3.2 Configuration Parameters

None.

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

None.

4.3 Demo

```
// Create the UMLModelManager and Diagram
UMLModelManager umlMgr = UMLModelManager.getInstance();
        Diagram diagram = new Diagram();
        diagram.setName("Main Class Diagram");
        umlMgr.addDiagram(diagram);


        // Assume we have two jar named "some_path/Test1.jar" and "some_path/Test2.jar",
        // it uses base exception 2.0 component. Base exception 2.0 has the path
        // "some_path/base_exception.jar". So we need to initialize the DefaultJarImporter
        // with the classpath to base exception 2.0
        JarImporter importer = new DefaultJarImporter(umlMgr,
        new String[]{"some_path/base_exception.jar"});

        // you also can create DefaultJarImporter with URLs
        importer = new DefaultJarImporter(umlMgr,
        new URL[]{new File("some_path/base_exception.jar").toURI().toURL()});

        // only import the JARs to Model
        importer.importJars(new String[]{"some_path/Test1.jar", "some_path/Test2.jar"});

        // you also can import JARs with specified URLs to Model
        importer.importJars(new URL[]{new File("some_path/Test1.jar").toURI().toURL()});

        // import the JARs to model, and then import all the entities into specified
        // diagram. Here assume there is a class diagram named with "Main Class Diagram"
        importer.importJarsToDiagram(new String[]{"some_path/Test1.jar",
        "some_path/Test2.jar"}, "Main Class Diagram");

        // you also can use URLs instead.
importer.importJarsToDiagram(
        new URL[]{new File("some_path/Test1.jar").toURI().toURL()},
        "Main Class Diagram");
```

5. Future Enhancement

Better support for various Java properties will be added in the future, for things like attributes applied to classes and operations to be shown as stereotypes, as well as direct Java source input, and possibly documentation as well.