UML Model Core 1.0 Component Specification

1. Design

The Unified Modeling Language (UML hereafter) is a language for specifying, visualizing, constructing and documenting the artifacts of software systems, as well as for business modeling. The UML is a standard maintained by the OMG (Object Management Group), and the version 1.4.2 has been adopted by ISO (reference ISO/IEC 19501:2005).

A model is a simplified, abstract (or actual) representation of a system from a particular point of view. Models are used in countless contexts; software engineering is one of them. Speaking in a strictly formal manner, UML defines a *metamodel*, a model that can be used to define other models; as stated above, the main target of UML are software-intensive systems and business processes, even if it can be used in a wide variety of other systems.

The UML metamodel contains a set of *metaclasses* (classes whose instances are classes theirselves) that are grouped in *packages*, whose members are logically homogeneous to manage language complexity. The “Foundation” package specifies the static structure of models, that is, it contains the metaclasses whose instances are the fundamental entities building up UML models (attributes, methods, and so on). The “Core” subpackage defines the very basic abstract and concrete metamodel constructs needed for the development of object models.

This component is a consistent set of Java classes that represent the great part of the metaclasses contained in the “Core” package of the UML 1.5 Specification. Not all metaclasses and attributes of the “Core” package are covered in this design; the less used features are left to be implemented by a future version of the component. This component is part of a greater group of components, the UML Model, that provides representations of the great part of the UML metamodel.

This component, along with the rest of the UML Model “macrocomponent”, can be easily used in a UML Tool that uses the MVC (Model-View-Controller) architecture pattern. In particular, this component is intended to be used mainly in the TopCoder UML Tool, so it follows its Architecture requirements and documentation.

1.1 Industry Standards

UML 1.5

1.2 Design Patterns

No design pattern is used in this design.

1.3 Required Algorithms

Obviously, this component doesn't contain any algorithm. As stated in the Design section, this component is intented to be only a *representation* of the Core part of the UML metamodel. The main target of this component is to be included in a MVC application, so all the actions that are needed to manage the UML model should be implemented in the Controller.

1.4 Component Class Overview

Interfaces

The following interfaces represent the metaclasses specified in the UML metamodel, package Foundation/Core.

Element

An element is an atomic constituent of a model.
In the metamodel, an Element is the top metaclass in the metaclass hierarchy. It has two subclasses: ModelElement and PresentationElement.

ModelElement

A model element is an element that is an abstraction drawn from the system being modeled. Contrast with view element, which is an element whose purpose is to provide a presentation of information for human comprehension.

In the metamodel, a ModelElement is a named entity in a Model. It is the base for all modeling metaclasses in the UML. All other modeling metaclasses are either direct or indirect subclasses of ModelElement.

Feature

A feature is a property, like operation or attribute, which is encapsulated within a Classifier.

In the metamodel, a Feature declares a behavioral or structural characteristic of an Instance of a Classifier or of the Classifier itself.

Namespace

A namespace is a part of a model that contains a set of ModelElements each of whose names designates a unique element within the namespace.

In the metamodel, a Namespace is a ModelElement that can own other ModelElements, like Associations and Classifiers. The name of each owned ModelElement must be unique within the Namespace. Moreover, each contained ModelElement is owned by at most one Namespace. The concrete subclasses of Namespace have additional constraints on which kind of elements may be contained.

GeneralizableElement

A generalizable element is a model element that may participate in a generalization relationship.

In the metamodel, a GeneralizableElement can be a generalization of other GeneralizableElements

Parameter

A parameter is an unbound variable that can be changed, passed, or returned. A parameter may include a name, type, and direction of communication. Parameters are used in the specification of operations, messages and events, templates, etc.

In the metamodel, a Parameter is a declaration of an argument to be passed to, or returned from, an Operation, a Signal, etc.

StructuralFeature

A structural feature refers to a static feature of a model element, such as an attribute. In the metamodel, a StructuralFeature declares a structural aspect of an Instance of a Classifier, such as an Attribute. For example, it specifies the multiplicity and changeability of the StructuralFeature. StructuralFeature is an abstract metaclass.

BehavioralFeature

A behavioral feature refers to a dynamic feature of a model element, such as an operation or method.

In the metamodel, a BehavioralFeature specifies a behavioral aspect of a Classifier. All different kinds of behavioral aspects of a Classifier, such as Operation and Method, are subclasses of BehavioralFeature.

Classifier

A classifier is an element that describes behavioral and structural features; it comes in several specific forms, including class, data type, interface, component, artifact, and others that are defined in other metamodel packages.

In the metamodel, a Classifier declares a collection of Features, such as Attributes, Methods, and Operations. It has a name, which is unique in the Namespace enclosing the Classifier.

Attribute

An attribute is a named slot within a classifier that describes a range of values that instances of the classifier may hold.

In the metamodel, an Attribute is a named piece of the declared state of a Classifier, particularly the range of values that Instances of the Classifier may hold.

Operation

An operation is a service that can be requested from an object to effect behavior. An operation

has a signature, which describes the actual parameters that are possible (including possible return values).
In the metamodel, an Operation is a BehavioralFeature that can be applied to the Instances of the Classifier that contains the Operation.

Method

A method is the implementation of an operation. It specifies the algorithm or procedure that effects the results of an operation.
In the metamodel, a Method is a declaration of a named piece of behavior in a Classifier and realizes one (directly) or a set (indirectly) of Operations of the Classifier.

Classes

The following are the classes included in the component. Each one of them (except ClassifierAbstractImpl) follows the same pattern: the class named <Name>Impl or <Name>AbstractImpl is a simple, java.util – based implementation of the <Name> interface (java.util – based means it uses concrete collection classes defined in that package: ArrayList for unordered collections and LinkedList for ordered collections). Any method in any class don't require more documentation or details than the one included in the corresponding interface.

ElementAbstractImpl
ModelElementAbstractImpl
FeatureAbstractImpl
NamespaceAbstractImpl
GeneralizableElementAbstractImpl
ParameterImpl
StructuralFeatureAbstractImpl
BehavioralFeatureAbstractImpl
AttributeImpl
OperationImpl
MethodImpl

ClassifierAbstractImpl

It is, like the other classes, a simple implementation of the Classifier interface based on the java.util concrete collection classes. However, some entities defined in the UML metamodel make use of multiple inheritance, and Classifier is one of them: it is a subclass of both Namespace and GeneralizableElement. Obviously ClassifierAbstractImpl cannot be subclass of both NamespaceAbstractImpl and GeneralizableAbstractImpl, so it just inherits the second one and reimplements all the methods of the first one.

1.5 Component Exception Definitions

This component doesn't declare any custom exception, since it doesn't contain any non-trivial code. That is, every concrete (i.e. implemented) method in this component simply sets or gets an attribute, or calls a corresponding method in an attribute collection.

Anyway, some error checking is done in certain methods. Non-collection attributes can be set to any value (either null) so setters and getters don't perform any error checking; however, null references can't be inserted in a collection attribute (either ordered or unordered), so an IllegalArgumentException is thrown in the case. Additionally, when accessing to a member of an ordered collection by mean of the index, the validity of such an index is checked, and an IndexOutOfBoundsException is thrown if it is below zero or greater or equal than the number of elements in the collection.

1.6 Thread Safety

The component is not thread safe.

This component's purpose is to represent (part of) an UML Model, and its main use is to be included in a UML Tool using MVC, as the Model part. It's very unlikely for such an application to require complex synchronization techniques: in the common use situations, it doesn't even need synchronization (i.e. it is a single-threaded application) or it has got one writing thread and multiple reading threads (for example in a GUI application). In the latter case, synchronization could (and should) be accomplished

externally to this component, considering the model as a whole entity to preserve consistency. So, this component doesn't provide synchronization at all to avoid performance hits.

2. Environment Requirements

2.1 Environment

- Development language: Java 1.5
- Compile target: Java 1.5

2.2 TopCoder Software Components

This component depends from a set of other components of the UML Model macrocomponent.

- UML Model - Datatypes 1.0
- UML Model - Core Extension Mechanisms 1.0
- UML Model - Core Dependencies 1.0
- UML Model - Core Auxiliary Elements 1.0
- UML Model - Model Management 1.0
- UML Model - State Machines 1.0
- UML Model - Core Relationships 1.0
- UML Model - Acitivity Graph 1.0
- UML Model - Common Behavior 1.0

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.

NOTE: The default location for 3rd party packages is ../lib relative to this component installation. Setting the ext_libdir property in topcoder_global.properties will overwrite this default location.

3. Installation and Configuration

3.1 Package Name

com.topcoder.uml.model.core

3.2 Configuration Parameters

None.

3.3 Dependencies Configuration

- Install all the TCS components listed in section 2.2, TopCoder Software Components.

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

```
/**
     * This class implements a Core interface, make it can save all AssociationEnds and Expressions.
     */
    private class RegisteringAttribute extends StructuralFeatureAbstractImpl implements
Attribute {

        /**
         * The value set.
         */
        private final Collection<Expression> values = new HashSet<Expression>();

        /**
         * The end set.
         */
        private final Collection<AssociationEnd> ends = new HashSet<AssociationEnd>();

        /**
         * Holds the initial value.
         */
        private Expression initialValue;

        /**
         * Holds the association end.
         */
        private AssociationEnd associationEnd;

        /**
         * The default constructor.
         */
        public RegisteringAttribute() {
        }

        /**
         * Sets the initial value.
         *
         * @param initialValue
         *            the initial value to set
         */
        public void setInitialValue(Expression initialValue) {
            if (initialValue != null) {
                values.add(initialValue);
            }
            this.initialValue = initialValue;
        }

        /**
         * Gets the initial value.
         *
         * @return the initial value
         */
        public Expression getInitialValue() {
            return initialValue;
        }

        /**
         * Sets the association end.
         *
         * @param associationEnd
         *            the association end to set
         */
        public void setAssociationEnd(AssociationEnd associationEnd) {
            if (associationEnd != null) {
                ends.add(associationEnd);
            }
            this.associationEnd = associationEnd;
        }

        /**
         * Gets the association end.
         *
         * @return the association end
         */
        public AssociationEnd getAssociationEnd() {
            return associationEnd;
        }

        /**
```

```
     * Gets all the initial values.
     *
     * @return all the initial values
     */
    public Collection<Expression> getAllInitialValues() {
        return new HashSet<Expression>(values);
    }

    /**
     * Gets all the association ends.
     *
     * @return all the association ends
     */
    public Collection<AssociationEnd> getAllAssociationEnds() {
        return new HashSet<AssociationEnd>(ends);
    }
}

/**
 * This class extends a Core class, it can't be created.
 */
private class ImmutableNamespace extends NamespaceAbstractImpl {

    /**
     * Creates an <code>ImmutableNamespace</code> instance.
     *
     * @param elements
     *            the elements the <code>ImmutableNamespace</code> instance will hold
     */
    public ImmutableNamespace(Collection<ModelElement> elements) {
        Iterator<ModelElement> iter = elements.iterator();
        while (iter.hasNext()) {
            addOwnedElement(iter.next());
        }
    }

    /**
     * Adds owned element.
     *
     * @param ownedElement
     *            the owned element to add
     */
    public void addOwnedElement(ModelElement ownedElement) {
        throw new UnsupportedOperationException("Addition of elements not supported");
    }

    /**
     * Removes owned element.
     *
     * @param ownedElement
     *            the owned element to removed
     * @return true if removed, otherwise false
     */
    public boolean removeOwnedElement(ModelElement ownedElement) {
        throw new UnsupportedOperationException("Removal of elements not supported");
    }

    /**
     * Clears owned elements.
     */
    public void clearOwnedElements() {
        throw new UnsupportedOperationException("Removal of elements not supported");
    }
}

/**
 * This test case show the demo usage of this component.
 */
public void testDemo() {
    // Create ImmutableNamespace object, should throw UnsupportedOperationException
    try {
        Collection<ModelElement> elements = new HashSet<ModelElement>();
        elements.add(new ModelElementImpl());
        new ImmutableNamespace(elements);
        fail("UnsupportedOperationException should be thrown.");
    } catch (UnsupportedOperationException e) {
        // success
    }

    RegisteringAttribute rAttribute = new RegisteringAttribute();
```

```
        rAttribute.setAssociationEnd(new AssociationEndImpl());
        rAttribute.setAssociationEnd(new AssociationEndImpl());
        System.out.println(rAttribute.getAllAssociationEnds().size());
        // Should be "2";

        // Instantiate Core objects
        Namespace com = new NamespaceImpl();
        Namespace topcoder = new NamespaceImpl();
        Namespace puzzle = new NamespaceImpl();
        ModelElement classesMdl = new ModelElementImpl();
        ModelElement usecasesMdl = new ModelElementImpl();
        Parameter retValue = new ParameterImpl();
        Operation addItems = new OperationImpl();

        // Set core objects non-collection attributes
        classesMdl.setName("Puzzle Classes");
        usecasesMdl.setName("Puzzle Use Cases");
        retValue.setKind(ParameterDirectionKind.RETURN);
        addItems.setConcurrency(CallConcurrencyKind.SEQUENTIAL);
        addItems.setLeaf(true);
        addItems.setAbstract(false);

        // Manipulate core objects' collections
        System.out.println(addItems.countMethods()); // Should output "0";

        com.addOwnedElement(topcoder);
        topcoder.addOwnedElement(puzzle);
        puzzle.addOwnedElement(classesMdl);
        puzzle.addOwnedElement(usecasesMdl);
        System.out.println(puzzle.countOwnedElements()); // Should print "2"

        Iterator<ModelElement> iter = puzzle.getOwnedElements().iterator();
        while (iter.hasNext()) {
            ModelElement e = iter.next();
            System.out.println(e.getName());
        }
        // This cycle should print "Puzzle Classes" and "Puzzle Use Cases".
        // The order is unspecified, since the collection of owned elements in Namespace is
unordered.

        System.out.println(puzzle.containsOwnedElement(classesMdl));
        // Should output "true"

        System.out.println(puzzle.containsOwnedElement(com));
        // Should output "false", since "com" namespace is not contained in package "puzzle".

        puzzle.clearOwnedElements();
        System.out.println(puzzle.countOwnedElements());
        // Should output "0"

        topcoder.removeOwnedElement(puzzle);
        System.out.println(topcoder.countOwnedElements());
        // Should output "0", since "puzzle" was the only element in "topcoder" package

        // Manipulate core objects' ordered collections
        Classifier classifier = new ClassifierImpl();
        Feature featureA = new FeatureImpl();
        Feature featureB = new FeatureImpl();

        System.out.println(classifier.indexOfFeature(featureA));
        // Should print "-1"

        classifier.addFeature(featureA);
        System.out.println(classifier.indexOfFeature(featureA));
        // Should print "0"

        classifier.setFeature(0, featureB);
        System.out.println(classifier.countFeatures());
        // Should print "1", since feature B replaced A
    }
}
```

5. Future Enhancements

Actually, this component doesn't cover all the classes and relationships defined in the Core package of the UML 1.5 Specification. A future enhancement could be to complete the implementation of the metamodel.

The component could be updated to follow UML 2.0 Specification.