UML Model – Data Types 1.0 Component Specification

1. Design

The UML Model - Data Types component declares the interfaces from the UML 1.5 framework, from the Data Types package. It provides concrete implementations for each interface and provides powerful API to access the collection attributes.

1.1 Design Patterns

None

1.2 Industry Standards

UML 1.5

1.3 Required Algorithms

There are no complex algorithms in this design.

1.4 Component Class Overview

AggregationKind
An enumeration that denotes what kind of aggregation an Association is. When placed on a target end, specifies the relationship of the target end to the source end.

ParameterDirectionKind
In the metamodel, ParameterDirectionKind defines an enumeration that denotes if a Parameter is used for supplying an argument and/or for returning a value.

CallConcurrencyKind
An enumeration that denotes the semantics of multiple concurrent calls to the same passive instance (i.e., an Instance originating from a Classifier with isActive=false).

PseudostateKind
In the metamodel, PseudostateKind defines an enumeration that discriminates the kind of Pseudostate..

ChangeableKind
In the metamodel, ChangeableKind defines an enumeration that denotes how an AttributeLink or LinkEnd may be modified.

ScopeKind
In the metamodel, ScopeKind defines an enumeration that denotes whether a feature belongs to individual instances or an entire classifier.

OrderingKind
Defines an enumeration that specifies how the elements of a set are arranged. Used in conjunction with elements that have a multiplicity in cases when the

multiplicity value is greater than one. The ordering must be determined and maintained by operations that modify the set. The intent is that the set of enumeration literals be open for new values to be added by tools for purposes of design, code generation, etc. For example, a value of sorted might be used for a design specification.

VisibilityKind
In the metamodel, VisibilityKind defines an enumeration that denotes how the element to which it refers is seen outside the enclosing name space.

Multiplicity
Simple, base interface. In the metamodel, a Multiplicity defines a non-empty set of non-negative integers. A set which only contains zero ({0}) is not considered a valid Multiplicity. Every Multiplicity has at least one corresponding String representation.

MultiplicityImpl
This is a simple, concrete implementation of Multiplicity interface.

MultiplicityRange
Simple, base interface. In the metamodel, a MultiplicityRange defines a range of integers. The upper bound of the range cannot be below the lower bound. The lower bound must be a nonnegative integer. The upper bound must be a nonnegative integer or the special value unlimited, which indicates there is no upper bound on the range.

MultiplicityRangeImpl
This is a simple, concrete implementation of MultiplicityRange interface.

Expression
Simple, base interface. In the metamodel, an Expression defines a statement which will evaluate to a (possibly empty) set of instances when executed in a context. An Expression does not modify the environment in which it is evaluated. An expression contains an expression string and the name of an interpretation language with which to evaluate the string.

ExpressionImpl
This is a simple, concrete implementation of Expression interface.

BooleanExpression
This interface extends Expression interface. In the metamodel, BooleanExpression defines a statement that will evaluate to an instance of Boolean when it is evaluated.

BooleanExpressionImpl
This is a simple concrete implementation of BooleanExpression interface and

extends ExpressionImpl. As such, all methods in BooleanExpression are supported.

MappingExpression
This interface extends Expression interface. An expression that evaluates to a mapping.

MappingExpressionImpl
This is a simple concrete implementation of MappingExpression interface and extends ExpressionImpl. As such, all methods in MappingExpression are supported.

ProcedureExpression
This interface extends Expression interface. In the metamodel, ProcedureExpression defines a statement that will result in a change to the values of its environment when it is evaluated.

ProcedureExpressionImpl
This is a simple concrete implementation of ProcedureExpression interface and extends ExpressionImpl. As such, all methods in ProcedureExpression are supported.

TypeExpression
This interface extends Expression interface. In the metamodel, TypeExpression is the encoding of a programming language type in the interpretation language. It is used within a ProgrammingLanguageDataType.

TypeExpressionImpl
This is a simple concrete implementation of TypeExpression interface and extends ExpressionImpl. As such, all methods in TypeExpression are supported.

1.5 Component Exception Definitions

This component defines no custom exceptions.

The general approach to parameter handling is not to do it. The architectural decision was to allow the beans to hold any state, and delegate to the users of these beans to decide what is legal and when it is legal.

1.6 Thread Safety

This component is not thread-safe, and there is no requirement for it to be thread-safe. In fact, the PM discourages method synchronization. Thread safety will be provided by the application using these implementations.

The classes are made non-thread-safe by the presence of mutable members and collections. In order to provide thread-safety, if that is ever desired, all simple member accessors and collections would need to be synchronized.

2. Environment Requirements

2.1 Environment

JDK 1.5

2.2 TopCoder Software Components

- TC UML Common Behavior 1.0
  - TC UML component defining the Common Behavior.

2.3 Third Party Components

None

3. Installation and Configuration

3.1 Package Names

com.topcoder.uml.model.datatypes
com.topcoder.uml.model.datatypes.expressions

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

The demo will demonstrate the usage of these beans. It will show them being instantiated, then used via their interface. This will be the typical usage of such simple entities under any scenario. This demo will focus on showing how a simple and collection attribute is managed, with the understanding that all other attributes are managed in exactly the same manner, and therefore not shown here.

4.3.1 Instantiation

```
// 1. Create instance of Multiplicity;
Multiplicity multiplicity = new MultiplicityImpl();

// 2. Create instance of MultiplicityRange;
MultiplicityRange multiplicityRange = new MultiplicityRangeImpl();
```

```
// 3. Create instance of Expression;
Expression expression = new ExpressionImpl();

// 4. Create instance of BooleanExpression;
BooleanExpression booleanExpression = new BooleanExpressionImpl();

// 5. Create instance of MappingExpression;
MappingExpression mappingExpression = new MappingExpressionImpl();

// 6. Create instance of ProcedureExpression;
ProcedureExpression procedureExpression = new ProcedureExpressionImpl();

// 7. Create instance of TypeExpression;
TypeExpression typeExpression = new TypeExpressionImpl();
```

4.3.2 Simple attributes management of Mutiplicity

```
// Create sample entity with a simple attribute to manage
Multiplicity multiplicity = new MultiplicityImpl();

// 1. addRange
MultiplicityRange range1 = new MultiplicityRangeImpl();
MultiplicityRange range2 = new MultiplicityRangeImpl();

// add range1
multiplicity.addRange(range1);
// add range1, then we have two range1's in this multiplicity entity
multiplicity.addRange(range1);

//5. containsRange
boolean containsRange1 = multiplicity.containsRange(range1);
// containsRange1 should be true, since we have added it in
boolean containsRange2 = multiplicity.containsRange(range2);
// containsRange2 should be false, since we have never added it

// 4. countRanges
// we have 2 ranges, so value of count1 equals 2.
int count1 = multiplicity.countRanges();

// 2. removeRange
boolean removed1 = multiplicity.removeRange(range2);
// since range2 does not exist, do nothing and return false

boolean removed2 = multiplicity.removeRange(range1);
// since range1 exists in the collection, remove one of the two range1's
// now, we have only one range1 in the collection and the countRanges returns
// one

// 5. containsRange
containsRange1 = multiplicity.containsRange(range1);
// containsRange1 should be true, since we still have one range1 object in
// it

// 3. getRanges
Collection<MultiplicityRange> retrievedRanges = multiplicity.getRanges();
// the collection constains only one range1 now
```

4.3.3 Simple attributes management of MutiplicityRange.

```
// Create sample entity with a simple attribute to manage
MultiplicityRange multiplicityRange = new MultiplicityRangeImpl();

// 1. Setter/Getter of Lower property
// lower should be non-negative
int lower = 3;
multiplicityRange.setLower(lower);
int retrievedLower = multiplicityRange.getLower();

// 2. Setter/Getter of Upper property
// upper should be non-negative
int upper = 30;
multiplicityRange.setUpper(upper);
int retrievedUpper = multiplicityRange.getUpper();

// 3. Setter/Getter of Muliplicity property
// multiplicity could be null
Multiplicity multiplicity = new MultiplicityImpl();
multiplicityRange.setMultiplicity(multiplicity);
Multiplicity retrievedMultiplicity = multiplicityRange.getMultiplicity();
```

4.3.4 *Simple attributes management of Expression.*

```
// Create sample entity with a simple attribute to manage
Expression expression = new ExpressionImpl();

// 1. Setter/Getter of Procedure property
// Use setter
Procedure procedure = new ProcedureImpl();
expression.setProcedure(procedure);
// Use getter
Procedure retrievedProcedure = expression.getProcedure();

// 2. Setter/Getter of Body property
// Use setter
String body = "23*3-(3-2)/1";
expression.setBody(body);
// Use getter
String retrievedBody = expression.getBody();

// 3. Setter/Getter of Language property
// Use setter
String language = "java";
expression.setLanguage(language);
// Use getter
String retrievedLanguage = expression.getLanguage();
```

4.3.5 *Simple attributes management of BooleanExpression.*

```
// Create sample entity with a simple attribute to manage
BooleanExpression expression = new BooleanExpressionImpl();

// 1. Setter/Getter of Procedure property
// Use setter
Procedure procedure = new ProcedureImpl();
expression.setProcedure(procedure);
// Use getter
Procedure retrievedProcedure = expression.getProcedure();

// 2. Setter/Getter of Body property
// Use setter
String body = "2 == 3";
expression.setBody(body);
// Use getter
String retrievedBody = expression.getBody();

// 3. Setter/Getter of Language property
// Use setter
String language = "java";
expression.setLanguage(language);
// Use getter
String retrievedLanguage = expression.getLanguage();
```

4.3.6 *Simple attributes management of MappingExpression.*

```
// Create sample entity with a simple attribute to manage
MappingExpression expression = new MappingExpressionImpl();

// 1. Setter/Getter of Procedure property
// Use setter
Procedure procedure = new ProcedureImpl();
expression.setProcedure(procedure);
// Use getter
Procedure retrievedProcedure = expression.getProcedure();

// 2. Setter/Getter of Body property
// Use setter
String body = "2 == 3";
expression.setBody(body);
// Use getter
String retrievedBody = expression.getBody();

// 3. Setter/Getter of Language property
// Use setter
String language = "java";
expression.setLanguage(language);
// Use getter
String retrievedLanguage = expression.getLanguage();
```

4.3.7 *Simple attributes management of ProcedureExpression.*

```
// Create sample entity with a simple attribute to manage
ProcedureExpression expression = new ProcedureExpressionImpl();

// 1. Setter/Getter of Procedure property
// Use setter
Procedure procedure = new ProcedureImpl();
expression.setProcedure(procedure);
// Use getter
Procedure retrievedProcedure = expression.getProcedure();

// 2. Setter/Getter of Body property
// Use setter
String body = "void hello(){}";
expression.setBody(body);
// Use getter
String retrievedBody = expression.getBody();

// 3. Setter/Getter of Language property
// Use setter
String language = "java";
```

```
expression.setLanguage(language);
// Use getter
String retrievedLanguage = expression.getLanguage();
```

4.3.8 Simple attributes management of TypeExpression.

```
// Create sample entity with a simple attribute to manage
TypeExpression expression = new TypeExpressionImpl();

// 1. Setter/Getter of Procedure property
// Use setter
Procedure procedure = new ProcedureImpl();
expression.setProcedure(procedure);
// Use getter
Procedure retrievedProcedure = expression.getProcedure();

// 2. Setter/Getter of Body property
// Use setter
String body = "public class A {}";
expression.setBody(body);
// Use getter
String retrievedBody = expression.getBody();

// 3. Setter/Getter of Language property
// Use setter
String language = "java";
expression.setLanguage(language);
// Use getter
String retrievedLanguage = expression.getLanguage();
```

4.3.9 Collection attribute management of Multiplicity.

```
// Create sample entity with a collection attribute to manage
Multiplicity multiplicity = new MultiplicityImpl();

// Use single-entity add method
MultiplicityRange ran1 = new MultiplicityRangeImpl();
multiplicity.addRange(ran1);
// There is now one range in the collection: {0}

// Use multiple-entity add method
Collection<MultiplicityRange> col1 = new ArrayList<MultiplicityRange>();
// add another five different ranges, {1, 2, 3, 4, 5}
MultiplicityRange[] colArray1 = new MultiplicityRange[5];
for (int i = 0; i < colArray1.length; i++) {
   colArray1[i] = new MultiplicityRangeImpl();
   col1.add(colArray1[i]);
}
// now col1: {1, 2 ,3, 4, 5}
// add duplicate ranges to col1
for (int i = 0; i < 3; i++) {
   col1.add(colArray1[1]);
}
// now col1 : {1, 2, 3, 4, 5, 1, 2, 3}
multiplicity.addRanges(col1);
// There will now be 9 ranges in the collection: {0, 1, 2, 3, 4, 5, 1, 2, 3}


// Use contains method to check for range presence
boolean present = multiplicity.containsRange(ran1);
// This will be true
// Use count method to get the number of ranges
int count = multiplicity.countRanges();
// The count will be 9

// Use single-entity remove method
boolean removed = multiplicity.removeRange(ran1);
// This will be true, and the collection size is 8, regardless
// if ran1 has duplicates in this collection.
// now collection: {1, 2, 3, 4, 5, 1, 2, 3}

// Use multiple-entity remove method
Collection<MultiplicityRange> col2 = new ArrayList<MultiplicityRange>();
// add three ranges to col2, such that col2 is a subset of col1
for (int i = 0; i < 3; i++) {
   col2.add(colArray1[i]);
}
// now col2: {1, 2, 3}
boolean altered = multiplicity.removeRanges(col2);
// This will be true, and the collection size is 5
// the collection will be  {4, 5, 1, 2, 3}, since the removeRanges is first-basis
// remove

// Use clear method
multiplicity.clearRanges();
// The collection size is 0 and contains no ranges
// now collection: {}
```

5. Future Enhancements

Providing a complete model, or moving to UML 2.