UML Model – Core Classifiers 1.0 Component Specification

1. Design

The UML Model - Core Classifiers component declares the interfaces from the UML 1.5 framework, from the Core – Classifiers 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

Class
This interface extends Classifier interface. The Classifier interface comes from the Core Requirements component. A class is a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. A class may use a set of interfaces to specify collections of operations it provides to its environment. In the metamodel, a Class describes a set of Objects sharing a collection of Features, including Operations, Attributes, and Methods that are common to the set of Objects. Furthermore, a Class may realize zero or more Interfaces; this means that its full descriptor must contain every Operation from every realized Interface (it may contain additional operations as well).

ClassImpl
This is a simple concrete implementation of Class interface and extends ClassifierAbstractImpl from the Core Requirements component. As such, all methods in Class are supported.

Interface
This interface extends Classifier interface. The Classifier interface comes from the Core Requirements component. An interface is a named set of operations that characterize the behavior of an element. In the metamodel, an Interface contains a set of Operations that together define a service offered by a Classifier realizing the Interface. A Classifier may offer several services, which means that it may realize several Interfaces, and several Classifiers may realize the same Interface.

InterfaceImpl
This is a simple concrete implementation of Interface interface and extends ClassifierAbstractImpl from the Core Requirements component. As such, all methods in Interface are supported.

DataType
This interface extends Classifier interface. The Classifier interface comes from the Core Requirements component. A data type is a type whose values have no identity (i.e., they are pure values). Data types include primitive built-in types (such as integer and string) as well as definable enumeration types (such as the predefined enumeration type boolean whose literals are false and true). In the metamodel, a DataType defines a special kind of Classifier in which Operations are all pure functions (i.e., they can return DataValues but they cannot change DataValues, because they have no identity). For example, an "add" operation on a number with another number as an argument yields a third number as a result; the target and argument are unchanged.

DataTypeImpl
This is a simple concrete implementation of DataType interface and extends ClassifierAbstractImpl from the Core Requirements component. As such, all methods in DataType are supported.

EnumerationLiteral
This interface extends ModelElement interface. The ModelElement interface comes from the Core Requirements component. An EnumerationLiteral defines an element of the run-time extension of an Enumeration data type. It has no relevant substructure, that is, it is atomic. The enumeration literals of a particular Enumeration datatype are ordered. It has a name (inherited from ModelElement) that can be used to identify it within its enumeration dataype.

EnumerationLiteralImpl
This is a simple concrete implementation of EnumerationLiteral interface and extends ModelElementAbstractImpl from the Core Requirements component. As such, all methods in EnumerationLiteral are supported.

Primitive
This interface extends DataType interface. A Primitive defines a predefined DataType, without any relevant UML substructure (i.e., it has no UML parts). A primitive datatype may have an algebra and operations defined outside of UML, for example, mathematically. Primitive datatypes used in UML itself include Integer, UnlimitedInteger, and String.

PrimitiveImpl
This is a simple concrete implementation of Primitive interface and extends DataTypeImpl. As such, all methods in Primitive are supported.

ProgrammingLanguageDataType
This interface extends DataType interface. A data type is a type whose values have no identity (i.e., they are pure values). A programming language data type is a data type specified according to the semantics of a particular programming language, using constructs available in that language. There are a wide variety of

programming languages and many of them include type constructs not included as UML classifiers. In some cases, it is important to represent those constructs such that their exact form in the programming language is available. The ProgrammingLanguagedData type captures such programming language types in a language-dependent fashion. They are represented by the name of the language and a string characterizing them, subject to interpretation by the particular language. Because they are dependent on particular languages, they are not portable among languages (except by agreement among the languages) and they do not map into other UML classifiers. Their semantics is therefore opaque within UML except by special interpretation by a profile intended for the particular language.

ProgrammingLanguageDataTypeImpl
This is a simple concrete implementation of ProgrammingLanguageDataType interface and extends DataTypeImpl. As such, all methods in ProgrammingLanguageDataType are supported.

Enumeration
This interface extends DataType interface. In the metamodel, Enumeration defines a kind of DataType whose range is a list of predefined values, called enumeration literals. Enumeration literals can be copied, stored as values, and passed as arguments. They are ordered within their enumeration datatype. An enumeration literal can be compared for an exact match or to a range within its enumeration datatype. There is no other algebra defined on them (e.g., they cannot be added or subtracted).

EnumerationImpl
This is a simple concrete implementation of Enumeration interface and extends DataTypeImpl. As such, all methods in Enumeration 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. The exception here is the list attributes. They will not allow null elements to be passed, and they will restrict the index to be valid for the state of that list.

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 Core Requirements 1.0
    - TC UML component defining the Core Requirements.
- TC UML Data Types 1.0
    - TC UML component defining the Data Types.

2.3 Third Party Components

None

3. Installation and Configuration

3.1 Package Names

com.topcoder.uml.model.core.classifiers

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 list 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

Create an instance of sample entity: `EnumerationLiteral`. All other concrete entities are instantiated in this manner and are not shown here.

```
// Create an instance of sample entity
EnumerationLiteral enumerationLiteral = new EnumerationLiteralImpl();
```

4.3.2 *Simple attribute management*

Manage a simple attribute: `EnumerationLiteral.enumeration`. All other simple attributes are managed in this manner and are not shown here.

```
// Create sample entity with a simple attribute to manage
EnumerationLiteral enumerationLiteral = an instance of
EnumerationLiteralImpl

// Use setter
Enumeration enumeration = some valid value
enumerationLiteral.setEnumeration(enumeration);

// Use getter
Enumeration retrievedEnumeration = enumerationLiteral.getEnumeration();
```

4.3.3 *List attribute management*

Manage a list attribute: `Enumeration.literals`.

```
// Create sample entity with a list attribute to manage
Enumeration enumeration = an instance of EnumerationImpl

// Use single-entity add method
EnumerationLiteral lit1 = some valid literal
enumeration.addLiteral(lit1);
// There is now one literal in the list

// Use multiple-entity add method
Collection<EnumerationLiteral> col1 = collection with 5 valid literals
enumeration.addLiterals(col1);
// There will now be 6 literals in the list

// Use single-entity, indexed add method, using lit1 again
enumeration.addLiteral(2,lit1);
// There are now 7 literals in the list, with
// another lit1 in third spot

// Use multiple-entity, indexed add method
Collection<EnumerationLiteral> col2 = collection with 2 valid literals
enumeration.addLiterals(3, col2);
// There will now be 9 literals in the list, with these two
// literals in fourth and fifth spots

// Use contains method to check for literal presence
boolean present = enumeration.containsLiteral(lit1);
// This will be true. It will locate the lit1 reference
// in the first spot.

// Use count method to get the number of literals
int count = enumeration.countLiterals();
// The count will be 9. Duplicates are counted as separate entities.

// Use single-entity remove method
boolean removed = enumeration.removeLiteral(lit1);
// This will be true, and the list size is 8, regardless
// if lit1 has duplicates in this list, which it does, and these
// are not removed

// Use multiple-entity remove method, using above col2
boolean altered = enumeration.removeLiterals(col2);
// This will be true, and the list size is 6

// Use clear method
enumeration.clearLiterals();
// The list size is 0 and contains no literals
```

5. Future Enhancements

Providing a complete model, or moving to UML 2.