UML Model – Common Behavior 1.0 Component Specification

1. Design

The UML Model - Common Behavior component declares the interfaces from the UML 1.5 framework, from the Common Behavior 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

Procedure
This interface extends ModelElement interface. The ModelElement interface comes form the Core Requirements component. A procedure is a coordinated set of actions that models a computation, such as an algorithm. It can also be used without actions to express a procedure in a textual language. In the metamodel, Procedure is a subclass of ModelElement. It can be linked to a Method or Expression to model how the method is carried out or the expression is evaluated.

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

Instance
This interface extends ModelElement interface. The ModelElement interface comes form the Core Requirements component. The instance construct defines an entity to which a set of operations can be applied and which has a state that stores the effects of the operations. In the metamodel, Instance is connected to at least one Classifier that declares its structure and behavior. It has a set of attribute values and is connected to a set of Links, both sets matching the definitions of its Classifiers. The two sets implement the current state of the Instance. An Instance may also own other Instances or Links.

InstanceAbstractImpl
This is a simple implementation of Instance interface and extends ModelElementAbstractImpl from the Core Requirements component. As such, all methods in Instance are supported. This class is declared as being abstract.

Stimulus

This interface extends ModelElement interface. The ModelElement interface comes form the Core Requirements component. A stimulus reifies a communication between two instances. In the metamodel, Stimulus is a communication (i.e., a Signal sent to an Instance, or an invocation of an Operation). It can also be a request to create an Instance, or to destroy an Instance. It has a sender, a receiver, and may have a set of actual arguments, all being Instances.

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

Object
This interface extends Instance interface. An object is an instance that originates from a class. In the metamodel, Object is a subclass of Instance and it originates from at least one Class. The set of Classes may be modified dynamically, which means that the set of features of the Object may change during its life-time.

ObjectImpl
This is a simple concrete implementation of Object interface and extends InstanceAbstractImpl. As such, all methods in Object are supported.

Link
This interface extends ModelElement interface. The ModelElement interface comes form the Core Requirements component. The link construct is a connection between instances. In the metamodel, Link is an instance of an Association. It has a set of LinkEnds that matches the set of AssociationEnds of the Association. A Link defines a connection between Instances.

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

LinkEnd
This interface extends ModelElement interface. The ModelElement interface comes form the Core Requirements component. A link end is an end point of a link. In the metamodel, LinkEnd is the part of a Link that connects to an Instance. It corresponds to an AssociationEnd of the Link’s Association.

LinkEndImpl
This is a simple concrete implementation of LinkEnd interface and extends ModelElementAbstractImpl from the Core Requirements component. As such, all methods in LinkEnd 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 collection and list attributes. They will not allow null elements to be passed, and for lists, it 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 Actions 1.0
  - TC UML component defining the Actions.
- 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.commonbehavior.procedure
com.topcoder.uml.model.commonbehavior.instances
com.topcoder.uml.model.commonbehavior.links

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, collection, 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: `Procedure`. All other concrete entities are instantiated in this manner and are not shown here.

```
// Create an instance of sample entity
Procedure procedure = new ProcedureImpl();
```

4.3.2 Simple attribute management

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

```
// Create sample entity with a simple attribute to manage
Procedure procedure = an instance of ProcedureImpl

// Use setter
Expression expression = some valid value
procedure.setExpression(expression);

// Use getter
Expression retrievedExpression = procedure.getExpression();
```

4.3.3 Collection attribute management

Manage a collection attribute: `Link.stimuli`. All other collection attributes are managed in this manner and are not shown here.

```
// Create sample entity with a collection attribute to manage
Link link = an instance of LinkImpl

// Use single-entity add method
Stimulus stim1 = some valid stimulus
link.addStimulus(stim1);
// There is now one stimulus in the collection
// Use multiple-entity add method
Collection<Stimulus> col1 = collection with 5 valid stimuli
link.addStimuli(col1);
// There will now be 6 stimuli in the collection

// Use contains method to check for stimulus presence
boolean present = link.containsStimulus(stim1);
// This will be true

// Use count method to get the number of stimuli
int count = link.countStimuli();
// The count will be 6

// Use single-entity remove method
boolean removed = link.removeStimulus(stim1);
// This will be true, and the collection size is 5, regardless
// if stim1 has duplicates in this collection.

// Use multiple-entity remove method
Collection<Stimulus> col2 = collection with 3 valid stimuli, which is a subset
of col1.
boolean altered = link.removeStimuli(col2);
// This will be true, and the collection size is 2

// Use clear method
link.clearStimuli();
// The collection size is 0 and contains no stimuli
```

4.3.4 *List attribute management*

Manage a list attribute: `Link.connections`.

```
// Create sample entity with a list attribute to manage
Link link = an instance of LinkImpl

// Use single-entity add method
LinkEnd conn1 = some valid connection
link.addConnection(conn1);
// There is now one connection in the list

// Use multiple-entity add method
Collection<LinkEnd> col1 = collection with 5 valid connections
link.addConnections(col1);
// There will now be 6 connections in the list

// Use single-entity, indexed add method, using conn1 again
link.addConnection(2,conn1);
// There are now 7 connections in the list, with
// another conn1 in third spot

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

// Use contains method to check for connection presence
boolean present = link.containsConnection(conn1);
// This will be true. It will locate the conn1 reference
// in the first spot.
// Use count method to get the number of connections
int count = link.countConnections();
// The count will be 9. Duplicates are counted as separate entities.

// Use single-entity remove method
boolean removed = link.removeConnection(conn1);
// This will be true, and the list size is 8, regardless
// if conn1 has duplicates in this list, which it does, and these
// are not removed

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

// Use clear method
link.clearConnections();
// The list size is 0 and contains no connections
```

5. Future Enhancements

Providing a complete model, or moving to UML 2.