UML Model Use Cases 1.0 Component Specification

1. Design

1.1 Overview

The UML Model Use Cases component provides interfaces and implementations to support the Use Cases activity graph functionality of the TopCoder UML tool. The four interfaces, Actor, UseCase, Include and Extend, all have concrete implementation classes. Additional methods were added to the Actor and UseCase implementation classes to simplify some of the API. The Extend hierarchy also includes the ‘condition’ attribute, which is part of the UML 1.5 specification.

1.2 Design Patterns

None.

1.3 Industry Standards

UML 1.5

1.4 Required Algorithms

There were no required algorithms in the requirements, however there is one slightly complex sequence in the UseCaseUtil utility class. The input to this method is a Classifier (either a UseCase or an Actor) and the output is the collection of Classifiers which are at the other end of the association, with the appropriate type. So, in the ActorImpl class, we would call it with UseCase.class as the ‘endType’, and in the UseCaseImpl class, we would call it with the Actor.class as the ‘endType’ argument.

```
Collection<Classifier> findAssociatedEnds(Classifier source, Class endType)
        // inputends is a collection of AssociationEnds
        inputends = source.getAssociations()
        for each e in inputends:
                // note, connection is a collection of type AssociationEnd
                connections = e.getAssocation().getConnections()
                for each connection c in connections
                        p = c.getParticipant()
                        if endType.isAssignableFrom(p.getClass())
                                outputends.add(p)
        return outputends
```

1.5 Component Class Overview

interface Actor extends Classifier

This interface represents an Actor in the UML Modeling tool. Aside from extending the Classifier interface, is an additional method, getUseCases, which traverses the various relationships attached to the Classifier that this interface extends, to find the UseCases that this Actor can invoke. Implementations of this interface should not be made thread-safe, since thread safety should be handled by the application itself.

class ActorImpl extends ClassifierAbstractImpl implements Actor

Implements the Actor interface. Aside from extending the ClassifierAbstractImpl class, there is an additional method, getUseCases, which traverses the various relationships attached to the Classifier that this interface extends, to find the UseCases that this Actor can invoke. It uses the UseCaseUtil method "findAssociatedEnd" to find the Actors. This class is required to be not thread-safe, since thread safety is required to be handled by the application itself.

interface UseCase extends Classifier

This interface represents a Use Case in the UML Modeling tool. Aside from extending the Classifier interface, it supports four relationship types: 1. Includes, where this UseCase includes another; 2. Includers where another UseCase includes this one; 3. Extends, where this UseCase is the extension ("extension extends base"); 4. Extenders where this UseCase is the base (is extended by another). In addition, there is an additional method, getActors, which traverses the various relationships attached to the Classifier that this interface extends, to find the Actors that can invoke this UseCase. Implementations of this interface should not be made thread-safe, since thread safety should be handled by the application itself.

class UseCaseImpl extends ClassifierAbstractImpl implements UseCase

This is the basic implementation of the UseCase interface, representing a Use Case in the UML Modeling tool. Aside from extending the ClassifierAbstractImpl class, it supports four relationship types: 1. Includes, where this UseCase includes another; 2. Includers where another UseCase includes this one; 3. Extends, where this UseCase is the extension ("extension extends base"); 4. Extenders where this UseCase is the base (is extended by another). In addition, there is an additional method, getActors, which traverses the various relationships attached to the Classifier that this interface extends, to find the Actors that can invoke this UseCase. It uses the UseCaseUtil method "findAssociatedEnd" to find the Actors. This class is required to be not thread-safe, since thread safety is required to be handled by the application itself.

class UseCaseUtil

This utility class provides the “findAssociatedEnd” method used by both the ActorImpl and UseCaseImpl classes. They use this to find their corresponding ”associated end” (Actor can find UseCases and UseCase can find Actors.) This class is thread-safe since there is no data in the class.

interface Extend extends Relationship

This interface describes an "Extends" relationship between two Use Cases. Therefore this interface captures the "base" use case as well as the UseCase that extends it, the "extension" UseCase. ("extension extends base") Implementations of this interface should not be made thread-safe, since thread safety should be handled by the application itself.

interface Include extends Relationship

This interface describes an "Include" relationship between two Use Cases. Therefore this interface captures the "base" use case as well as the UseCase that it includes, the "addition" UseCase. (“base includes addition”) Implementations of this interface should not be made thread-safe, since thread safety should be handled by the application itself.

abstract class UseCaseRelationshipAbstractImpl extends RelationshipAbstractImpl

This abstract class provides the basic implementation of both the Include and the Extend interfaces, as it stores both a base UseCase and a "target" (either the 'included' or the 'extended' UseCase). It will be the base class for the IncludeImpl and ExtendImpl concrete classes. It extends the RelationshipAbstractImpl class so it implements the Relationship interface as well. This class is required to be not thread-safe, since thread safety is required to be handled by the application itself.

class ExtendImpl extends UseCaseRelationshipAbstractImpl implements Extend

This class implements the Extend interface. It uses its base class, UseCaseRelationshipAbstractImpl, to provide the get/setBase functionality; the get/setExtension methods are delegated to the get/setTarget methods of the superclass. This class is required to be not thread-safe, since thread safety is required to be handled by the application itself.

class IncludeImpl extends UseCaseRelationshipAbstractImpl implements Include

This class implements the Include interface. It uses its base class, UseCaseRelationshipAbstractImpl, to provide the get/setBase functionality; the get/setAddition methods are delegated to the get/setTarget methods of the superclass. This class is required to be not thread-safe, since thread safety is required to be handled by the application itself.

1.6 Component Exception Definitions

None

1.7 Thread Safety

The classes in this component are not thread-safe, since thread-safety is not required. Instead, the calling application is required to ensure thread safety.

2. Environment Requirements

2.1 Environment

- At minimum, Java 1.5 is required for compilation and executing test cases.

2.2 TopCoder Software Components

- UML Model Core Requirements 1.0. The UseCase and Actor interfaces and their corresponding implementation classes use the Classifier interface and its corresponding implementation classes from this component.
- UML Model Core Relationships 1.0. The Extend and Include interfaces and their corresponding implementation classes use the Relationship interface and its corresponding implementation classes from this component.
- UML Model Data Types 1.0. The BooleanExpression interface in this component is referenced by the Extends relationship.

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

3. Installation and Configuration

3.1 Package Name

`com.topcoder.uml.model.usecases`

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.
- Execute ‘ant test’ within the directory that the distribution was extracted to.

4.2 Required steps to use the component

- Extract the component distribution.
- Follow the demo.

4.3 Demo

Note: this demo (and in fact this component) uses many interfaces and classes in other UML Model components that may not yet be completed. The developer is required to mock up both the interface and the implementation for testing purposes only but should not be judged on the quality of those mock classes as long as they provide the base functionality needed to implement the component and this demo.

4.3.1 Manage Actors

```
// Create an empty actor
Actor actor = new ActorImpl();
// This is a method in the ModelElement interface, which is one of the base
// interfaces of Actor
actor.setName("Application");
System.out.println(actor.getName());

// Should be empty initially
Collection<UseCase> associated = actor.getUseCases();

// Add a UseCase related to the Actor – notice how many steps it takes
// set up the use-case side of the association
UseCase useCase1 = new UseCaseImpl();
useCase1.setName("Manage Actors");
AssociationEnd useCaseEnd = new AssociationEndImpl();
useCaseEnd.setParticipant(useCase1);
actor.addAssociation(useCaseEnd);

// set up the actor side of the association
AssociationEnd actorEnd = new AssociationEndImpl();
actorEnd.setParticipant(actor);
useCase1.addAssociation(actorEnd);

// connect the ends up in an Association
Association assoc = new AssociationImpl();
assoc.addConnection(useCaseEnd);
assoc.addConnection(actorEnd);
useCaseEnd.setAssociation(assoc);
actorEnd.setAssociation(assoc);

// Should have the UseCase we added through the lengthy process above (useCase1)
Collection<UseCase> useCases = actor.getUseCases();
System.out.println(useCases);
```

4.3.2 Manage UseCases

```
UseCase useCase2 = new UseCaseImpl();
// This is a method in the ModelElement interface, which is one of the base
// interfaces of UseCase
useCase2.setName("Manage Use Cases");
System.out.println(useCase2.getName());

// Should be empty initially
Collection<Actor> associated = useCase2.getActors();

// Add an Actor related to the UseCase – notice how many steps it takes
// set up the use-case side of the association
Association assoc = new AssociationImpl();
AssociationEnd useCaseEnd = new AssociationEndImpl();
```

```
useCaseEnd.setParticipant(useCase2);

// set up the actor side of the association
Actor actor = new ActorImpl();
actor.addAssociation(useCaseEnd);
actor.setName("Application");
AssociationEnd actorEnd = new AssociationEndImpl();
actorEnd.setParticipant(actor);
useCase2.addAssociation(actorEnd);

assoc.addConnection(useCaseEnd);
assoc.addConnection(actorEnd);
useCaseEnd.setAssociation(assoc);
actorEnd.setAssociation(assoc);

// Should include the Actor we added through the lengthy process above.
Collection<Actor> actors = useCase2.getActors();
System.out.println(actors);
```

4.3.3 Create Include relationships

```
base = new UseCaseImpl();
base.setName("Manage Include relationships of a UseCase");
addition = new UseCaseImpl();
addition.setName("Do something more");

include = new IncludeImpl(base, addition);
// we can get the base and addition
System.out.println(include.getBase());
System.out.println(include.getAddition());

addition2 = new UseCaseImpl();
addition2.setName("Do something else");

include2 = new IncludeImpl();
include2.setBase(base);
include2.setAddition(addition2);
```

4.3.4 Manage Include relationships of a UseCase

This demo section uses the variables defined in Section 4.3.3

```
// create the variables firstly
testIncludeCreationDemo();

// should have zero includes initially
System.out.println(base.countIncludes());

// add the includes
base.addInclude(include);
base.addInclude(include2);

// see if it included the first include object
System.out.println(base.containsInclude(include));
//         another way to do it – use the UseCase
System.out.println(base.containsInclude(addition2));

// get the include objects – should have include and include2
Collection<Include> includes = base.getIncludes();
//      another way to get included use cases – should have addition and addition2
Collection<UseCase> included = base.getIncludedUseCases();

// should have two includes
```

```
System.out.println(base.countIncludes());

// remove one (should return true)
base.removeInclude(include);
// should be one now
System.out.println(base.countIncludes());
// try to remove it again (should return false and do nothing else)
base.removeInclude(include);
// should still be one
System.out.println(base.countIncludes());

// remove "all" of them
base.clearIncludes();
// should have none
System.out.println(base.countIncludes());
```

4.3.5 Manage Includer relationships of a UseCase

This demo section uses the variables defined in Section 4.3.3

```
// create the variables firstly
testIncludeCreationDemo();

// should have zero initially
System.out.println(base.countIncluders());

// add the include
addition.addIncluder(include);

// see if it included the include object
System.out.println(addition.containsIncluder(include));
// another way to do it – use the UseCase
System.out.println(addition.containsIncluder(base));

// get the include objects
Collection<Include> includers = addition.getIncluders();
// another way to get including use cases
Collection<UseCase> including = addition.getIncludingUseCases();

// should have one includer
System.out.println(base.countIncluders());

// remove it
addition.removeIncluder(include);
// should be none now
System.out.println(addition.countIncluders());

// add the include again, for testing purposes
addition.addIncluder(include);
addition.clearIncluders();
// should have none
System.out.println(addition.countIncluders());
```

4.3.6 Create Extend relationships

```
base = new UseCaseImpl();
base.setName("Create relationship");
base.setAbstract(true);

extension = new UseCaseImpl();
extension.setName("Create Extend relationships");

extend = new ExtendImpl(base, extension);
System.out.println(extend.getBase());
```

```
        System.out.println(extend.getExtension());

        extension2 = new UseCaseImpl();
        extension2.setName("Create Include relationships");
        extend2 = new ExtendImpl();
        extend2.setBase(base);
        extend2.setExtension(extension2);
```

4.3.7 Manage Extend relationships of a UseCase

This demo section uses the variables defined in Section 4.3.6

```
        // create the variables firstly
        testExtensionCreationDemo();

        // should have zero initially
        System.out.println(extension.countExtends());

        // add the extension
        extension.addExtend(extend);

        // should have one extends
        System.out.println(extension.countExtends());

        // see if it contains the extension using the Extend object
        System.out.println(extension.containsExtend(extend));
        // another way to do it – use the UseCase
        System.out.println(extension.containsExtend(base));

        // get the extend objects
        Collection<Extend> extendObjects = extension.getExtends();
        // another way to get extended use cases
        Collection<UseCase> bases = extension.getBaseUseCases();

        // remove the extension
        extension.removeExtend(extend);
        // should be none now
        System.out.println(extension.countExtends());

        // re-add the extension
        extension.addExtend(extend);
        extension.clearExtends();
        // should have none
        System.out.println(extension.countExtends());
```

4.3.8 Manage Extender relationships of a UseCase

This demo section uses the variables defined in Section 4.3.6

```
        // create the variables firstly
        testExtensionCreationDemo();

        // should have zero initially
        System.out.println(base.countExtenders());

        // add the extenders
        base.addExtender(extend);
        base.addExtender(extend2);

        // see if it includes the first extend object
        System.out.println(base.containsExtender(extend));
        System.out.println(base.containsExtender(extend2));
```

```
// another way to do it – use the UseCase
System.out.println(base.containsExtender(extension));
System.out.println(base.containsExtender(extension2));

// get the extend objects – should be 2
Collection<Extend> extenders = base.getExtenders();
// another way to get extension use cases – should be the 2 use cases
Collection<UseCase> subUseCases = base.getSubUseCases();

// should have two extenders
System.out.println(base.countExtenders());

// remove one
base.removeExtender(extend);
// should be one now
System.out.println(base.countExtenders());

base.clearExtenders();
// should have none
System.out.println(base.countExtenders());
```

5. Future Enhancements

- Provide the complete UML 1.5 model
- Provide additional helper methods to create and remove relationships between Actors and UseCases, and between UseCases. This was not done in this version because the structure of XMI files would probably result in a more automated (mechanical) approach to creating objects using Reflection. Any additional 'helper' methods would likely not be used.
- Provide factory methods to create Include or Extend relationships given the participating use cases. This was not done in this design for the same reasons as the previous enhancement.
- Support UML 2.0