UML Model – Core Dependencies 1.0 Component Specification

1. Design

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

Dependency
This interface extends Relationship interface. The Relationship interface comes from the Core Relationships component. A term of convenience for a Relationship other than Association, Generalization, Flow, or metarelationship (such as the relationship between a Classifier and one of its Instances). A dependency states that the implementation or functioning of one or more elements requires the presence of one or more other elements. In the metamodel, a Dependency is a directed relationship from a client (or clients) to a supplier (or suppliers) stating that the client is dependent on the supplier (i.e., the client element requires the presence and knowledge of the supplier element).

DependencyImpl
This is a simple concrete implementation of Dependency interface and extends RelationshipAbstractImpl from the Core Relationships component. As such, all methods in Dependency are supported.

Binding
This interface extends Dependency interface. A binding is a relationship between a template (as supplier) and a model element generated from the template (as client). It includes a list of arguments that match the template parameters. The template is a form that is cloned and modified by substitution to yield an implicit model fragment that behaves as if it were a direct part of the model. A Binding must have one supplier and one client; unlike a general Dependency, the supplier and client may not be sets. In the metamodel, a Binding is a Dependency where the supplier is the template and the client is the instantiation of the template that performs the substitution of parameters of a template. A Binding has a list of arguments that replace the parameters of the supplier to yield the client. The client is fully specified by the binding of the supplier's parameters and does not add any information of its own. An element may participate as a supplier in multiple

Binding relationships to different clients. An element may participate in only one Binding relationship as a client.

BindingImpl
This is a simple concrete implementation of Binding interface and extends DependencyImpl. As such, all methods in Binding are supported.

Abstraction
This interface extends Dependency interface. An abstraction is a Dependency relationship that relates two elements or sets of elements that represent the same concept at different levels of abstraction or from different viewpoints. In the metamodel, an Abstraction is a Dependency in which there is a mapping between the supplier and the client. Depending on the specific stereotype of Abstraction, the mapping may be formal or informal, and it may be unidirectional or bidirectional.

AbstractionImpl
This is a simple concrete implementation of Abstraction interface and extends DependencyImpl. As such, all methods in Abstraction are supported

Usage
This interface extends Dependency interface. A usage is a relationship in which one element requires another element (or set of elements) for its full implementation or operation. The relationship is not a mere historical artifact, but an ongoing need; therefore, two elements related by usage must be in the same model. In the metamodel, a Usage is a Dependency in which the client requires the presence of the supplier. How the client uses the supplier, such as a class calling an operation of another class, a method having an argument of another class, and a method from a class instantiating another class, is defined in the description of the particular Usage stereotype.

UsageImpl
This is a simple concrete implementation of Usage interface and extends DependencyImpl. As such, all methods in Usage 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 Core Requirements 1.0
    - TC UML component defining the Core Requirements.
- TC UML Core Relationships 1.0
    - TC UML component defining the Core Relationships.
- TC UML Core Auxiliary Elements 1.0
    - TC UML component defining the Core Auxiliary Elements.
- 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.dependencies

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

```
// Create an instance of sample entity
Abstraction abstraction = new AbstractionImpl();
```

4.3.2 Simple attribute management

Manage a simple attribute: `Abstraction.mapping`.

```
// Create an instance of Usage
Usage usage = new UsageImpl();

// Create an instance of Abstraction
Abstraction abstraction = new AbstractionImpl();

// Create an instance of MappingExpression
MappingExpression mapping = new MockMappingExpression();

// Use setter
abstraction.setMapping(mapping);
// Use getter
MappingExpression retrievedMapping = abstraction.getMapping();
```

4.3.3 Collection attribute management

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

```
// Create sample entity with a collection attribute to manage
Dependency dependency = new DependencyImpl();

// Use single-entity add method
ModelElement supp1 = new MockModelElement();
dependency.addSupplier(supp1);
// There is now one supplier in the collection
// Use multiple-entity add method
Collection<ModelElement> col1 = new ArrayList<ModelElement>();
ModelElement supp2 = new MockModelElement();
ModelElement supp3 = new MockModelElement();

// collection with 3 valid suppliers
col1.add(supp1);
col1.add(supp2);
col1.add(supp3);
dependency.addSuppliers(col1);
```

```
// There will now be 4 suppliers in the collection
// Use contains method to check for supplier presence
// This will be true
boolean present1 = dependency.containsSupplier(supp1);
boolean present2 = dependency.containsSupplier(supp2);
boolean present3 = dependency.containsSupplier(supp3);

// Use count method to get the number of suppliers
// The count will be 4
int count = dependency.countSuppliers();

// Get the Collection
Collection<ModelElement> collection = dependency.getSuppliers();

// Use single-entity remove method
// This will be true, and the collection size is 3
boolean removed = dependency.removeSupplier(supp1);


// if supp1 has duplicates in this collection.
// Use multiple-entity remove method
Collection<ModelElement> col2 = new ArrayList<ModelElement>();
col2.add(supp2);
col2.add(supp3);

// This will be true, and the collection size is 1
boolean altered = dependency.removeSuppliers(col2);

// Use clear method
// The collection size is 0 and contains no suppliers
dependency.clearSuppliers();


// Use single-entity add method
ModelElement client1 = new MockModelElement();
dependency.addClient(client1);
// There is now one supplier in the collection
// Use multiple-entity add method
Collection<ModelElement> col3 = new ArrayList<ModelElement>();
ModelElement client2 = new MockModelElement();
ModelElement client3 = new MockModelElement();

// collection with 3 valid suppliers
col3.add(client1);
col3.add(client2);
col3.add(client3);
dependency.addClients(col3);

// There will now be 4 suppliers in the collection
// Use contains method to check for supplier presence
// This will be true
present1 = dependency.containsClient(client1);
present2 = dependency.containsClient(client2);
present3 = dependency.containsClient(client3);

// Use count method to get the number of suppliers
// The count will be 4
count = dependency.countClients();

// Get the Collection
collection = dependency.getClients();
```

```
// Use single-entity remove method
// This will be true, and the collection size is 3
removed = dependency.removeClient(client1);


// if client1 has duplicates in this collection.
// Use multiple-entity remove method
Collection<ModelElement> col4 = new ArrayList<ModelElement>();
col4.add(client2);
col4.add(client3);

// This will be true, and the collection size is 1
altered = dependency.removeClients(col4);

// Use clear method
// The collection size is 0 and contains no suppliers
dependency.clearClients();
```

4.3.4 *List attribute management*

Manage a collection attribute: `Binding.arguments`.

```
// Create sample entity with a list attribute to manage
Binding binding = new BindingImpl();

// Use single-entity add method
TemplateArgument arg1 = new MockTemplateArgument();
// There is now one argument in the list
binding.addArgument(arg1);

// Use multiple-entity add method
Collection<TemplateArgument> col1 = new ArrayList<TemplateArgument>();
TemplateArgument arg2 = new MockTemplateArgument();
TemplateArgument arg3 = new MockTemplateArgument();
//collection with 3 valid arguments
col1.add(arg1);
col1.add(arg2);
col1.add(arg3);
// There will now be 4 arguments in the list
binding.addArguments(col1);

// Use single-entity, indexed add method, using arg1 again
binding.addArgument(2, arg1);
// There are now 5 arguments in the list, with another arg1 in third spot

// Use multiple-entity, indexed add method
Collection<TemplateArgument> col2 = new ArrayList<TemplateArgument>();
//collection with 2 valid arguments
col2.add(arg2);
col2.add(arg3);
// There will now be 7 arguments in the list, with these two
// arguments in fourth and fifth spots
binding.addArguments(3, col2);

// Set the Argument
binding.setArgument(5, arg1);

// Get the Collection
Collection<TemplateArgument> collection = binding.getArguments();

// Gets the index of the appearance of the TemplateArgument
int n = binding.indexOfArgument(arg2);
```

```
// Use contains method to check for argument presence
// This will be true. It will locate the arg1 reference in the first spot.
boolean present = binding.containsArgument(arg1);

// Use count method to get the number of arguments
// The count will be 7. Duplicates are counted as separate entities.
int count = binding.countArguments();

// Removes the TemplateArgument instance at the index position
TemplateArgument arg4 = binding.removeArgument(5);

// Use single-entity remove method
// This will be true, and the list size is 6, regardless if arg1 has
duplicates
// in this list, which it does, and these are not removed
boolean removed = binding.removeArgument(arg1);

// Use multiple-entity remove method, using above col2
// This will be true, and the list size is 4
boolean altered = binding.removeArguments(col2);

// Use clear method
// The list size is 0 and contains no arguments
binding.clearArguments();
```

5. Future Enhancements

Providing a complete model, or moving to UML 2.