UML Model – Model Management 1.0 Component Specification

1. Design

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

ElementImport
Simple, base interface. An element import defines the visibility and alias of a model element included in the namespace within a package, as a result of the package importing another package. In the metamodel, an ElementImport reifies the relationship between a Package and an imported ModelElement, concrete implementation of Serializable interface.

ElementImportImpl
This is a simple, concrete implementation of ElementImport interface.

Package
This interface extends Namespace and GeneralizableElement interfaces. The Namespace and GeneralizableElement interfaces come form the Core Requirements component. A package is a grouping of model elements. In the metamodel, Package is a subclass of Namespace and GeneralizableElement. A Package contains ModelElements like Packages, Classifiers, and Associations. A Package may also contain Constraints and Dependencies between ModelElements of the Package.

PackageImpl
This is a simple concrete implementation of Package interface and extends GeneralizableElementAbstractImpl from the Core Requirements component. To facilitate complete implementation of methods in the interface, the methods in Namespace interface are implemented but all they do is defer to in internal concrete implementation of that Namespace. In fact, it will be a simple inner concrete extension of NamespaceAbstractImpl, which also comes from the Core Requirements component. As such, all methods in Collaboration are supported.

Subsystem
This interface extends Package and Classifier interfaces. The Classifier interface comes form the Core Requirements component. A subsystem is a grouping of model elements that represents a behavioral unit in a physical system. A subsystem offers interfaces and has operations. In addition, the model elements of a subsystem are partitioned into specification and realization elements, where the former, together with the operations of the subsystem, are realized by the latter. In the metamodel, Subsystem is a subclass of both Package and Classifier.

As such it may have a set of Features, which are constrained to be Operations and Receptions, and Associations.

SubsystemImpl
This is a simple concrete implementation of Subsystem interface and extends ClassifierAbstractImpl from the Core Requirements component. To facilitate complete implementation of methods in the interface, the methods in Package interface are implemented but all they do is defer to in internal concrete implementation of that Package – PackageImpl. As such, all methods in Subsystem are supported.

Model
This interface extends Package interface. A model captures a view of a physical system. It is an abstraction of the physical system, with a certain purpose. This purpose determines what is to be included in the model and what is irrelevant. Thus the model completely describes those aspects of the physical system that are relevant to the purpose of the model, at the appropriate level of detail.

ModelImpl
This is a simple concrete implementation of Model interface and extends PackageImpl. As such, all methods in Model are supported.

NamespaceImpl
Inner class that represents a concrete extension of Namespace. It simply provides a concrete wrapper of the NamespaceAbstractImpl for this class so it can make use of the logic in the Namespace branch of implementation.

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 attributes. They will not allow null elements to be passed.

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.modelmanagement

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

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

```
// Create an instance of sample entity
ElementImport elementImport = new ElementImportImpl();
```

4.3.2 Simple attribute management

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

```
// Create sample entity with a simple attribute to manage
ElementImport elementImport = new ElementImportImpl();
// Use setter
Package aPackage = new PackageImpl();
elementImport.setPackage(aPackage);

// Use getter
Package retrievedPackage = elementImport.getPackage();
```

4.3.3 *Collection attribute management*

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

```
// Create sample entity with a collection attribute to manage
Package aPackage = new PackageImpl();

// Use single-entity add method
ElementImport imp1 = new ElementImportImpl();
aPackage.addElementImport(imp1);
// There is now one elementImport in the collection

// Use multiple-entity add method
Collection<ElementImport> col1 = new ArrayList<ElementImport>();
for (int i = 0; i < 5; i++) {
        col1.add(new ElementImportImpl());
 }
aPackage.addElementImports(col1);
// There will now be 6 elementImports in the collection

// Use contains method to check for elementImport presence
boolean present = aPackage.containsElementImport(imp1);
// This will be true

// Use count method to get the number of elementImports
int count = aPackage.countElementImports();
// The count will be 6

// Use single-entity remove method
boolean removed = aPackage.removeElementImport(imp1);
// This will be true, and the collection size is 5, regardless
// if imp1 has duplicates in this collection.

// Use multiple-entity remove method
Collection<ElementImport> col2 = new ArrayList<ElementImport>();
boolean altered = aPackage.removeElementImports(col2);
// This will be true, and the collection size is 2

// Use clear method
aPackage.clearElementImports();
// The collection size is 0 and contains no elementImports
```

5. Future Enhancements

Providing a complete model, or moving to UML 2.