Class Associations v1.0 Component Specification

1. Design

The Class Associations Component provides the user with a mechanism for associating a specific "Handler" Object with a specific "target" class. With this mechanism in place, the user can then retrieve the appropriate "Handlers" for instances of the "target" Class. The appropriate "Handler" is defined by the Association Algorithm that is used.

The default Association Algorithm simply returns the handler associated with the "target" class. If none are found, then the handler associated with the "closest" superclass of the target will be returned (see Required Algorithms for more details).

Limitations of the Component

The functionality of the component has some limitations, completely arising from the nature of the problem: This component attempts to solve the problem of determining which Object is the appropriate one to be used in a given situation. This implies that the User is unable to determine the Object at compile time (otherwise, the User would simply specify the object in the source code rather than use this component).

This means that the user will not know which methods to invoke upon the returned "Handler" object! The user will not be sure whether an instance of Class A (which has methodA) or an instance of Class B (which has methodB) would be returned, since this cannot be determined until run-time!

To address this problem, the Component must be utilized in three ways:

1. All handlers registered with a specific ClassAssociator must conform to a common interface. In section 1.3 of the Requirements Specification, the registered handlers are "RectangleDrawer" and "SquareDrawer" - this suggests that both handlers conform to the "Drawer" interface. Any returned Handlers can then be cast to the common interface and the client can work with the handlers via the API provided by that interface.

2. All handlers registered with a specific ClassAssociator must conform to a common class. Each handler instance can then be configured with different attribute parameters, so that their behavior would be different according to the "target" class. Similar to #1, the returned Handlers can be cast to the common class.

3. Use reflection to determine what methods can be invoked on the returned "Handler" object, then invoke the appropriate methods.

Of the three methods outlined above, methods #1, and #2 are the most likely to be used. Again, the nature of the problem limits it that the methods above are imposed "contractually", or with "usage warnings" - otherwise, the flexibility of the component is severely compromised (it would mean that the clients would have to design their objects with this component in mind - and this is unacceptable).

However, to help facilitate the correct use of the component via methods #1 and #2, the component provides methods that can limit the type of "Handlers" accepted to a specific class or interface. In this way, accidental misuse of the component (by adding the wrong Handler type) is avoided.

Handlers for "Target" Interfaces

The other major issue to consider here is that of "target" interfaces. The example in the Requirements Specification document only has "Handlers" that are registered with "Target" classes instead of with a "Target" interface.

Interfaces have the potential to cause collision problems: An object can belong to a class and implement one or more interfaces at the same time. Resolving a collision - ie, determining which is the "correct" handler to return - may be a problem.

In this case, the "correct" handler to return would be the one registered with the class, since a class is more specific than an interface. Only when no "Class Handlers" can be found will an "Interface Handler" be returned. The default would return the first "Interface Handler" that can be found that has no subclass "Interface Handlers" of the "Target".

This is albeit naive solution, but the existence of pluggable algorithms means that better methods for resolving Interface collision can be added, if and when the need arises.

1.1 Design Patterns

Future component direction indicate that additional association algorithms may be useful. As such, the **Strategy** pattern is applied in order to offer support of any future algorithm implementations.

1.2 Industry Standards

None.

1.3 Required Algorithms

DefaultAlgorithm

This is the algorithm that was specified in the Requirement's Specification:

- Find handler objects whose associated class is the type of the target, and return it if present.
- If there is no association with the actual type of the target, search for associations (that can handle subtypes) that are associated with supertypes of the target.
- Of these associations, search for one that is associated with a class that is a subtype of all others.

The above algorithm is pretty simple, and it can be implemented using 1-2 loops (depending on implementation), and Class.isAssignableFrom(Class c) method. The implementation should be very similar to an algorithm for finding the minimum number within a list of numbers.

However, the default algorithm in our case has the following additional conditions (these conditions apply to the handlers that are associated with supertypes of the target):

- If the handler is associated with a class that is an interface, this handler must NOT be returned unless there are no handlers associated with "true" or "non-interface" classes. Otherwise, return the handler associated with the Class.
- If two or more interface handlers are found, then return an interface handler such that there are no other interface handlers which are registered as subtypes of the returned interface handler. Any candidate is suitable.

The Class.isInterface() method can be very helpful in implementing the added conditions. The additional logic can be done by adding another extra pass over the list of classes. It is even possible to store the Interface associations during the initial pass to save even more time.

1.4 Component Class Overview

ClassAssociator:

This is the main class of the Class Associations component. The ClassAssociator encapsulates the mapping of "Handler" objects to specific "Target" classes. The ClassAssociator can then retrieve the appropriate "Handler" for a specified "Target" class or object. The "Handlers" are retrieved using the default Association Algorithm, which simply returns the handler which is "closest" to the class of the "Target" that was specified. The ClassAssociator also offers support for additional pluggable association algorithms.

AssociationAlgorithm *(Interface)*:

An object that can retrieve "Handlers" for specific targets. Each implementation offers different logic of determining which is the appropriate "Handler" to return.

DefaultAssociationAlgorithm:

Default implementation of the AssociationAlgorithm. This implementation would return the handler that is directly associated with the class of the "target". If no such handler is available, the handler of the "closest" class to the target is returned. (See Required Algorithms Section for more details)

1.5 Component Exception Definitions

IllegalArgumentException:

Thrown by nearly all the methods in this component when a null value is passed as a parameter. It is also thrown if the client attempts to remove an algorithm using DEFAULT_ALGORITHM as the specified algorithm name, since this would cause the ClassAssociator to break.

IllegalHandlerException:

The custom IllegalHandlerException is thrown when a "Handler" object is registered, that is not an instance of the specified RestrictionHandler class, and the client specifies that Handler Restriction must be enforced.

IllegalStateException:

Thrown by the setHandlerRestriction and setHandlerRestrictionClass methods when they specify to disallow a class of "Handlers" which are already present within the ClassAssociator.

1.6 Component Benchmark and Stress Tests

The component has a set of mapping routines as well as an Association Algorithm that at the worst runs at linear O(n) complexity.

Testing the performance of this component will consist of measuring the period of time it takes to perform the DefaultAlgorithm. To test the component, a ClassAssociator must be set up with 20, 40, and 60 Associations arranged in a linear hiearchy. The number of executions within 3 seconds of the retrieveHandler method will then be measured - the execution it takes to perform these operations must scale with linear inversion to the number of Associations within the ClassAssociator. (ie 20 Associations must perform ~3x more executions than 60 Associations).

It should also be noted that the first execution of this method will probably be slower than usual, because the JVM has to load the class. As such, it may also be advisable to have a "start-up" test before performing the actual tests.

The ClassAssociator is not thread-safe. However, several instances of the ClassAssociator can be initialized (one for each thread), and no significant performance overheads, except for the JVM thread management, should be expected.

2. Environment Requirements

2.1 Environment

- Development Language: Java 1.4
- Compile Targets: Java 1.3 and Java 1.4

2.2 TopCoder Software Components

No additional software components are required.

2.3 Third Party Components

No Third Party Components are required.

3. Installation and Configuration

3.1 Package Name

com.topcoder.util.classassociations

3.2 Configuration Parameters

None.

3.3 Dependencies Configuration

No dependencies are needed, so no configuration is required.

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

First, configure the component and make sure that it is accessible from the classpath.

Decide which family of "Handler" objects the ClassAssociator will return.

(Optional) Specify the class of the family of "Handler" objects with the setHandlerRestriction() and setHandlerRestrictionClass() methods, to make misuse more unlikely.

To Associate a Handler with a Single Class:

- Call ClassAssociator.addAssociation() with either the target class or an instance of the target class.

To Associate a Handler with a Hiearchy of Classes:

- Call ClassAssociator.addGroupAssociation() with either the target class or an instance of the target class hiearchy.

To Retrieve a Handler for a Class:

- Call ClassAssociator.retrieveHandler() with either the target class or an instance of the target Class.
- It would probably be useful to cast the returned Object to the class of the "Handler"

family of this ClassAssociator.

To Add a Custom Algorithm:

- Create your own class that implements the AssociationAlgorithm interface.
- Call ClassAssociator.addAlgorithm() with the name of the Algorithm and an instance of your custom class.

To Retrieve a Handler for a Class using a Custom Algorithm:

- Call ClassAssociator.retrieveHandler() with the target class, and the name of the Algorithm.

4.3 Demo

```
      // This code demonstrates the use of the Class Associations
      // Component.

      import com.topcoder.examples.*;

      import com.topcoder.util.classassociations.ClassAssociator;

      ClassAssociator ca = new ClassAssociator();

      // In this example, the family of "Handlers" will be implementing
      // the "Validator" interface.  Set HandlerRestriction to make the
      // ClassAssociator "safer" to use.

      ca.setHandlerRestrictionClass(Class.forName("com.topcoder.example
s.Validator"));

      ca.setHandlerRestriction(true);

      // Now, start assigning associations.

      try {

      ca.addAssociation(Class.forName("com.topcoder.examples.MediumData
"), new MediumValidator());

      ca.addAssociation(new SmallData(), new SmallValidator());

      ca.addGroupAssociation(new EnormousData(), new
EnormousValidator());

      } catch (IllegalHandlerException e) {

            System.err.println("The handlers that were specified do not
      implement the "Validator" interface!");

      }

      // At some later stage, we can then retrieve these associations

      RandomData foo = new RandomData();

      Validator fooHandler = (Validator) ca.retrieveHandler(foo);

      // Use fooHandler to your heart's content!

      fooHandler.validate(foo);
```

5. Future Enhancements

Implement additional AssociationAlgorithms. A notable issue is a better method of resolving collisions when a Target Class implements two interfaces, both of which have a registered Handler.