Resource Management 1.1 Component Specification

New items are marked with red and changed ones with blue.

1. Design

The Resource Management component provides resource management functionalities. A resource can be associated with a project, phase and a list of submissions. Each resource will have a role which identifies the resource's responsibilities for the associated scope. A set of resources can be created, updated or searched for a project. Notifications can also be assigned and unassigned to users. The persistence logic for resources and related entities is pluggable.

For development, this component is split into two parts. The main part will need to be done first, as the persistence part can not be done without the object model classes (in the main part) are complete.

- Main part: This development project will be responsible for all classes and interfaces in the com.topcoder.management.resource package and the com.topcoder.management.resource.search classes. The ResourcePersistenceException will also fall in this development project because it is declared to be thrown in the ResourceManager interface.
- Persistence part: This development project will be responsible for the classes in the com.topcoder.management.resource.persistence and com.topcoder.management.resource.persistence.sql packages. Development of the ResourcePersistenceException will not fall in this development project, as it will already have been done in the main part.

Version 1.1 adds the ability to associate multiple submissions to a resource. Only the Resource and ResourceFilterBuilder have been changed to accommodate the new requirements. The rest is exactly the same as in version 1.0.
Several sections of the initial Component Specification as well as some diagram objects have been removed because they were not relevant to the project and with all the updates they would have caused inconsistency problems.
The requirements asked for the addition of two methods and this design added 6 more plus a few Search filters factory methods. Given the extremely limited scope of this design they could be considered enhancements.

1.1 Design Patterns

The ResourcePersistence interface and implementations uses the **Strategy Pattern**, as does the ResourceManager interface and implementations. The Delegate pattern is used to delegate filter building tasks to the Search Builder component.

1.2 Industry Standards

SQL

1.3 Required Algorithms

None.

1.4 Component Class Overview

ResourceManager:

The ResourceManager interface provides the ability to persist, retrieve and search for persisted resource modeling objects. This interface provides a higher level of interaction than the ResourcePersistence interface. For example, the updateResource method will determine if the resource is new, and if so, assign it an id before persisting it, whereas the ResourcePersistence interface will simply fail in this situation. The methods in this interface break down into dealing with the 4 main modeling classes in this component, and the methods for each modeling class are quite similar.

Implementations of this interface are not required to be thread safe.

AuditableResourceStructure:

The AuditableResourceStructure is the base class for the modeling classes in this component. It holds the information about when the structure was created and updated. This class simply holds the four data fields needed for this auditing information and exposes both getters and setters for these fields.

All the methods in this class do is get/set the underlying fields, so this class will be very easy to develop.

This class is highly mutable. All fields can be changed.

Resource:

The Resource class is the main modeling class in this component. It represents any arbitrary resource. The Resource class is simply a container for a few basic data fields. All data fields in this class are mutable and have get and set methods.

The only thing to take note of when developing this class is that the setId method throws the IdAlreadySetException.

This class is highly mutable. All fields can be changed.

Changes in version 1.1: Multiple submissions can be associated with a Resource.

ResourceRole:

The ResourceRole class is the second modeling class in this component. It represents a type of resource and is used to tag instances of the Resource class as playing a certain role. The ResourceRole class is simply a container for a few basic data fields. All data fields in this class are mutable and have get and set methods.

The only thing to take note of when developing this class is that the setId method throws the IdAlreadySetException.

This class is highly mutable. All fields can be changed.

NotificationType:

The NotificationType class is the third modeling class in this component. It represents a type of notification. It is orthogonal to the Resource and ResourceRole classes. This class is simply a container for a few basic data fields. All data fields in this class are mutable and have get and set methods.

The only thing to take note of when developing this class is that the setId method throws the IdAlreadySetException.

This class is highly mutable. All fields can be changed.

Notification:

The Notification class is the final modeling class in this component. It represents a notification, which is an association between an external id (the use and meaning of this field is up to the user of the component), a project, and a notification type. This class is simply a container for these data fields. All data fields (directly in this class) are immutable and have only getters.

This class is mutable because its base class is mutable.

ResourceFilterBuilder:

The ResourceFilterBuilder class supports building filters for searching for Resources. This class consists of 2 parts. The first part consists of static Strings that name the fields that are available for searching. All ResourceManager implementations should use SearchBundles that are configured to use these names. The second part of this class consists of convenience methods to create common filters based on these field names. Development-wise, this class should prove quite simple, as all methods are a single line.

This class has only final static fields/methods, so mutability is not an issue.

Changes in version 1.1: Added a new filter factory method createAnySubmissionIdFilter(long[]) based on the fact that resource has been changed to be associated with one or many submission ids.

ResourceRoleFilterBuilder:

The ResourceRoleFilterBuilder class supports building filters for searching for ResourceRoles. This class consists of 2 parts. The first part consists of static Strings that name the fields that are available for searching. All ResourceManager implementations should use SearchBundles that are configured to use these names. The second part of this class consists of convenience methods

to create common filters based on these field names. Development-wise, this class should prove quite simple, as all methods are a single line.

This class has only final static fields/methods, so mutability is not an issue.

NotificationFilterBuilder:

The NotificationFilterBuilder class supports building filters for searching for Notifications. This class consists of 2 parts. The first part consists of static Strings that name the fields that are available for searching. All ResourceManager implementations should use SearchBundles that are configured to use these names. The second part of this class consists of convenience methods to create common filters based on these field names. Development-wise, this class should prove quite simple, as all methods are a single line.

This class has only final static fields/methods, so mutability is not an issue.

NotificationTypeFilterBuilder:

The NotificationTypeFilterBuilder class supports building filters for searching for NotificationTypes. This class consists of 2 parts. The first part consists of static Strings that name the fields that are available for searching. All ResourceManager implementations should use SearchBundles that are configured to use these names. The second part of this class consists of convenience methods to create common filters based on these field names. Development-wise, this class should prove quite simple, as all methods are a single line.

This class has only final static fields/methods, so mutability is not an issue.

ResourcePersistence:

The ResourcePersistence interface defines the methods for persisting and retrieving the object model in this component. This interface handles the persistence of the four classes that make up the object model – Resources, ResourceRoles, Notifications, and NotificationTypes. This interface is not responsible for searching the persistence for the various entities. This is instead handled by a ResourceManager implementation.

Implementations of this interface are not required to be thread-safe or immutable.

SqlResourcePersistence:

The SqlResourcePersistence class implements the ResourcePersistence interface, in order to persist to the database structure given in the resource_management.sql script. This class does not cache a Connection to the database. Instead a new Connection is used on every method call. Most methods in this class will just create and execute a single PreparedStatement. However, some of the Resource related methods will need to execute several

PreparedStatements in order to accomplish the update/insertion/deletion of the Resource.

This class is immutable and thread-safe in the sense that multiple threads can not corrupt its internal data structures. However, the results if used from multiple threads can be unpredictable as the database is changed from different threads. This can equally well occur when the component is used on multiple machines or multiple instances are used, so this is not a thread-safety concern.

PersistenceResourceManager:

The PersistenceResourceManager class implements the ResourceManager interface. It ties together a persistence mechanism, several Search Builder searching instances (for searching for various types of data), and several id generators (for generating ids for the various types). This class consists of several methods styles. The first method style just calls directly to a corresponding method of the persistence. The second method style first assigns values to some data fields of the object before calling a persistence method. The third type of method uses a SearchBundle to execute a search and then uses the persistence to load an object for each of the ids found from the search.

This class is immutable and hence thread-safe.

1.5 Component Exception Definitions

ResourcePersistenceException:

The ResourcePersistenceException indicates that there was an error accessing or updating a persisted resource store. This exception is used to wrap the internal error that occurs when accessing the persistence store. For example, in the SqlResourcePersistence implementation it is used to wrap SqlExceptions.

This exception is initially thrown in ResourcePersistence implementations and from there passes through ResourceManager implementations and back to the caller. It is also thrown directly by some ResourceManager implementations.

IdAlreadySetException:

The IdAlreadySetException is used to signal that the id of one of the resource modeling classes has already been set. This is used to prevent the id being changed once it has been set.

This exception is initially thrown in the 3 setId methods of the resource modeling classes.

1.6 Thread Safety

This component is not thread safe. This decision was made because the modeling classes in this component are mutable while the persistence classes make use of non-thread-safe components such as Search Builder. Combined with there being no business oriented requirement to make the component thread safe, making this

component thread safe would only increase development work and needed testing while decreasing runtime performance (because synchronization would be needed for various methods). These tradeoffs can not be justified given that there is no current business need for this component to be thread-safe.

This does not mean that making this component thread-safe would be particularly hard. Making the modeling classes thread safe can be done by simply adding the synchronized keyword to the various set and get methods. The persistence and manager classes would be harder to make thread safe. In addition to making manager and persistence methods synchronized, there would need to be logic added to handle conditions that occur when multiple threads are manipulating the persistence. For example, “Resource removed between SearchBuilder query and loadResource call”.

2. Environment Requirements

2.1 Environment

Java 1.4+ is required for compilation, testing, or use of this component

2.2 TopCoder Software Components

- Search Builder 1.3.2: Used for searching for resources and related resources.
- DB Connection Factory 1.0: Used in SQL persistence implementation to connect to the database. Also used by the Search Builder component.
- ID Generator 3.0: Used to create ids when new Resources and related objects are created.
- Configuration Manager 2.1.5: Used to configure the DB Connection Factory and Search Builder components. Can also be used with the Object Factory component. Not used directly in this component.
- Object Factory 2.0.1: Can be used to create ResourceManager implementations, particularly the PersistenceResourceManager class. There is no compile time or runtime dependency on this component.
- Base Exception 1.0: Used for being extended for custom exception classes.
- Database Abstraction 1.1: It is used by Search Builder component.
- Data Validation 1.0: It is used by Search Builder component.
- Class Associations 1.0: It is used by Search Builder component.

2.3 Third Party Components

None.

3. Installation and Configuration

3.1 Package Name

com.topcoder.management.resource
com.topcoder.management.resource.persistence
com.topcoder.management.resource.persistence.sql
com.topcoder.management.resource.search

3.2 Configuration Parameters

No direct configuration is used for this component. The Object Factory component can be used to create SqlResourcePersistences and PersistenceResourceManagers if desired. The SearchBundles and IDGenerators needed can be configured or created programmatically. For configuration of these objects, see the relevant component specifications.

When using the SQL database structure given in the resource_management.sql script, the SearchBundles passed to the PersistenceResourceManager should be configured to use the following contexts (queries minus where clause):

```
SELECT resource.resource_id
FROM resource
LEFT OUTER JOIN resource_submission
      ON resource.resource_id = resource_submission.resource_id
LEFT OUTER JOIN resource_info
      ON resource.resource_id = resource_info.resource_id
LEFT OUTER JOIN resource_info_type_lu
      ON resource_info.resource_info_type_id =
            resource_info_type_lu.resource_info_type_id
WHERE
```

For searching ResourceRoles

```
SELECT resource_role_id
FROM resource_role_lu
WHERE
```

For searching NotificationTypes

```
SELECT notification_type_id
FROM notification_type_lu
WHERE
```

For searching Notifications

```
SELECT external_ref_id, project_id, notification_type_id
FROM notification
WHERE
```

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

Install and configure the other TopCoder component following their instructions. Then follow section 4.1 and the demo.

4.3 Demo

Unfortunately, as this component can not really be described in a full customer scenario without including all the other components up for design this week, this demo will not be a customer oriented demo but a rundown of the requirements in a demo form. Where it makes sense, examples of what a customer might do with the information are shown.

4.3.1 Create a Resource and ResourceRole

```
ResourceRole resourceRole = new ResourceRole();
resourceRole.setName("Some Resource Role");
resourceRole.setDescription("This role plays some purpose");

// Note that it is not necessary to set any other field of the
// resource because they are all optional
Resource resource = new Resource();
resource.setResourceRole(resourceRole);

// The creation of notification types is entirely similar
// to the calls above.
```

4.3.2 Create persistence and manager

```
SqlResourcePersistence persistence =
      new SqlResourcePersistence(
            <connection factory loaded from configuration>)

ResourceManager manager = new PersistenceResourceManager(
      persistence,
      <search builders loaded from configuration: See Search
            Builder component for configuration details>,
      <id generators loaded from configuration: See Search
            Builder component for configuration details>);
```

4.3.3 Save the created Resource and ResourceRole to the persistence

```
// Note that this will assign an id to the resource and
// resource role
manager.updateResourceRole(resourceRole, "Operator #1");
manager.updateResource(resource, "Operator #1");
// The updating of notification types is entirely similar
```

```
// to the calls above

// The data can then be changed and the changes
// persisted
resourceRole.setName("Changed name");
manager.updateResourceRole(resourceRole, "Operator #1");
```

4.3.4 *Update All Resources For a Project*

```
long projectId = 1205;
// Removes any resources for the project not in the array
// and updates/adds those in the array to the persistence
manager.updateResources(new Resource[] {resource},
      projectId, “Operator #1”);
```

4.3.5 *Retrieve and search resources*

```
// Get a resource for a given id
Resource resource2 = manager.getResource(14402);
// The properties of the resource can then be queried
// and used by the client of this component

// Search for resources
// Build the filters – this example shows searching for
// all resources related to a given project and of a
// given type and having an extension property of a given name
Filter projectFilter =
      ResourceFilterBuilder.createProjectIdFilter(953);
Filter resourceTypeFilter =
      ResourceFilterBuilder.createResourceRoleIdFilter(1223);
Filter extensionNameFilter =
      ResourceFilterBuilder.createExtensionPropertyNameFilter(
            “Extension Prop Name”);
Filter fullFilter =
      SearchBundle.buildAndFilter(SearchBundle.buildAndFilter(
            projectFilter, resourceTypeFilter),
            extensionNameFilter);

// Search for the Resources
Resource[] matchingResources =
      manager.searchResources(fullFilter);

// ResourceRoles, NotificationTypes, and Notifications can be
// searched similarly by using the other FilterBuilder classes
// and the corresponding ResourceManager methods.  They can
// also be retrieved through the getAll methods
ResourceRole[] allResourceRoles = manager.getAllResourceRoles();
NotificationType[] allNotificationTypes =
      Manager.getAllNotificationTypes();
```

4.3.6 *Add/Remove notifications*

```
// Note that it is up to the application to decide what the
// user/external ids represent in their system
manager.addNotifications(new long[] {1, 2, 3}, 953, 192,
      “Operator #1”);
manager.removeNotifications(new long[] {1, 2, 3}, 953, 192,
      “Operator #1”);
```

4.3.7 *Retrieve notifications*

```
long[] users = manager.getNotifications(953, 192);
// This might, for example, represent the users to which an email
// needs to be sent.  The client could then lookup the user
// information for each id and send the email.

// Searches for notifications can also be made using an API
// that precisely parallels that shown for Resources in 4.3.5.
// When searching is used, full-fledges Notification instances
// are returned.
```

4.3.8 *Manipulating Resource submissions*

```
// Create a Resource
Resource resource = new Resource();

// Suppose we have some submissions that need to be associated
with the Resource created.
Long[] submissions =
             new Long[] {new Long(1), new Long(2), new Long(3)};

// Add the submissions to the Resource
resource.setSubmissions(submissions);

// Suppose a new submission is received
Long newSubmission = new Long(4);

resource.addSubmission(newSubmission);

// See whether a resource contains a submission
Long checkSubmission = new Long(3);


resource.containsSubmission(checkSubmission);

// See how many submissions are associated with the Resource
int submissionsNumber = resource.countSubmissions();


// Clear the associated submissions
resource.clearSubmissions();


// Create Resource filter which is used to retrieve the resources
// which are associated with given submission id.
Filter resourceFilter =
                ResourceFilterBuilder.createSubmissionIdFilter(2);

// Create Resource filter which is used to retrieve the resources
// which are associated with any one of given submission ids.
resourceFilter =
            ResourceFilterBuilder.createAnySubmissionIdFilter(
                                                  new long[]{1, 4});
```

5. Future Enhancements

At the current time, no future enhancements are expected for this component.