Deliverable Management 1.0 Component Specification

1. Design

The Deliverable Management component provides deliverable management functionalities. Various deliverables need to be fulfilled for a project during a specific phase. Usually a phase can be concluded only when all the required deliverables are present. The component defines an API to track the deliverables. The actual mechanism to verify each deliverable will be pluggable.

It also supports two types of specific deliverables, document upload and submission. Submission is a type of upload that has additional properties.

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) being complete.

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

1.1 Design Patterns

The DeliverablePersistence interface and implementations uses the **Strategy Pattern**, as do the DeliverableManager, UploadPersistence, and UploadManager interfaces and implementations.

1.2 Industry Standards

SQL

1.3 Required Algorithms

The only complicated part of this component is the SQL queries needed in the SqlUploadPersistence and SqlDeliverablePersistence classs. Beyond this, nothing more complicated than a simple for loop or an if/else is needed in this component.

1.3.1 Sql Queries for SqlUploadPersistence

This section lists the SQL queries and statements that will be needed by the SqlUploadPersistence class, accompanied by a written explanation of what to do, when more logic than just a sequence of SQL statements is needed. All of these SQL statements should be executed using PreparedStatements. A typical method of this class would look like (this example uses the "Update Upload Type" statement):

SQL statement to execute:

```
UPDATE upload_type_lu
SET name = ?, description = ?, modify_user = ?, modify_date = ?
WHERE upload_type_id = ?
```

Sample Code:

```
// Open connection
Connection connection = connectionFactory.createConnection();
// Create a prepared statement
PreparedStatement ps =
      connection.prepareStatement(< above text >);
// Use data in NotificationType passed to method to set
// parameters in prepared statement
ps.setString(1, uploadType.getName());
ps.setString(2, uploadType.getDescription());
ps.setString(3, uploadType.getModificationUser());
ps.setDate(4, uploadType.getModificationDate());
ps.setLong(5, uploadType.getId());
// execute PreparedStatement
ps.execute();
// close connection
connection.close();
```

1.3.1.1 Add Upload Type

```
INSERT INTO upload_type_lu
(upload_type_id, name, description, create_user, create_date,
modify_user, modify_date)
VALUES (?, ?, ?, ?, ?, ?, ?)
```

1.3.1.2 Remove Upload Type

```
DELETE FROM upload_type_lu
WHERE upload_type_id = ?
```

1.3.1.3 Update Upload Type

```
UPDATE upload_type_lu
SET name = ?, description = ?, modify_user = ?, modify_date = ?
WHERE upload_type_id = ?
```

1.3.1.4 Load Upload Type

```
SELECT upload_type_id, name, description, create_user,
create_date, modify_user, modify_date
```

```
FROM upload_type_lu
WHERE upload_type_id = ?
```

1.3.1.4.1 Load Upload Types

This is used to load multiple upload types with a single query. The only difference is that the WHERE clause becomes:

```
WHERE upload_type_id IN (<id_values>)
```

1.3.1.5 Add Upload Status

```
INSERT INTO upload_status_lu
(upload_status_id, name, description, create_user, create_date,
modify_user, modify_date)
VALUES (?, ?, ?, ?, ?, ?, ?)
```

1.3.1.6 Remove Upload Status

```
DELETE FROM upload_status_lu
WHERE upload_status_id = ?
```

1.3.1.7 Update Upload Status

```
UPDATE upload_status_lu
SET name = ?, description = ?, modify_user = ?, modify_date = ?
WHERE upload_status_id = ?
```

1.3.1.8 Load Upload Status

```
SELECT upload_status_id, name, description, create_user,
create_date, modify_user, modify_date
FROM upload_status_lu
WHERE upload_status_id = ?
```

1.3.1.8.1 Load Upload Statuses

This is used to load multiple upload statuses with a single query. The only difference is that the WHERE clause becomes:

```
WHERE upload_status_id IN (<id_values>)
```

1.3.1.9 Add Submission Status

```
INSERT INTO submission_status_lu
(submission_status_id, name, description, create_user,
create_date, modify_user, modify_date)
VALUES (?, ?, ?, ?, ?, ?, ?)
```

1.3.1.10 Remove Submission Status

```
DELETE FROM submission_status_lu
WHERE submission_status_id = ?
```

1.3.1.11 Update Submission Status

```
UPDATE submission_status_lu
SET name = ?, description = ?, modify_user = ?, modify_date = ?
WHERE submission_status_id = ?
```

1.3.1.12 Load Submission Status

```
SELECT submission_status_id, name, description, create_user,
create_date, modify_user, modify_date
FROM submission_status_id
WHERE submission_status_id = ?
```

1.3.1.12.1 Load Submission Statuses

This is used to load multiple submissions statuses with a single query. The only difference is that the WHERE clause becomes:

```
WHERE submission_status_id IN (<id_values>)
```

1.3.1.13 Add Upload

```
INSERT INTO upload
(upload_id, project_id, resource_id, upload_type_id,
upload_status_id, parameter, create_user, create_date,
modify_user, modify_date)
VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
```

1.3.1.14 Remove Upload

```
DELETE FROM upload
WHERE upload_id = ?
```

1.3.1.15 Update Upload

```
UPDATE upload
SET project_id = ?, resource_id = ?, upload_type_id = ?,
upload_status_id = ?, parameter = ?, modify_user = ?, modify_date
= ?
WHERE upload_id = ?
```

1.3.1.16 Load Upload

```
SELECT upload_id, project_id, resource_id, upload_type_id,
upload_status_id, parameter, upload.create_user,
upload.create_date, upload.modify_user, upload.modify_date,
upload_type_lu.name, upload_type_lu.upload_type_id,
upload_status_lu.name, upload_status.upload_status_id,
upload_type_lu.description, upload_status_lu.description,
upload_type_lu.create_user, upload_type_lu.create_date,
upload_type_lu.modify_user, upload_type_lu.modify_date,
upload_status_lu.create_user, upload_status_lu.create_date,
upload_status_lu.modify_user, upload_status_lu.modify_date
FROM upload
INNER JOIN upload_type_lu
      ON upload_type.upload_type_id = upload.upload_type_id
INNER JOIN upload_status_lu
      ON upload_status.upload_status_id = upload.upload_status_id
WHERE upload_id = ?
```

1.3.1.16.1 Load Uploads

This is used to load multiple uploadsd with a single query. All UploadType and UploadStatus objects should be loaded through the getAll methods, and these should be joined with the results of this query, instead of looking up each status/type object with a separate query. This will result in only three database queries being made, which will improve the performance of the application.

```
SELECT upload_id, project_id, resource_id, upload_type_id,
upload_status_id, parameter, upload.create_user,
upload.create_date, upload.modify_user, upload.modify_date,
upload_type_lu.name, upload_type_lu.upload_type_id,
upload_status_lu.name, upload_status.upload_status_id,
upload_type_lu.description, upload_status_lu.description,
upload_type_lu.create_user, upload_type_lu.create_date,
upload_type_lu.modify_user, upload_type_lu.modify_date,
upload_status_lu.create_user, upload_status_lu.create_date,
upload_status_lu.modify_user, upload_status_lu.modify_date
FROM upload
INNER JOIN upload_type_lu
      ON upload_type.upload_type_id = upload.upload_type_id
INNER JOIN upload_status_lu
      ON upload_status.upload_status_id = upload.upload_status_id
WHERE submission_id IN (<id_values>)
```

1.3.1.16.2

1.3.1.17 Add Submission

```
INSERT INTO submission
(submission_id, upload_id, submission_status_id, create_user,
create_date, modify_user, modify_date)
VALUES (?, ?, ?, ?, ?, ?, ?)
```

1.3.1.18 Remove Submission

```
DELETE FROM submission
WHERE submission_id = ?
```

1.3.1.19 Update Submission

```
UPDATE submission
SET upload_id = ?, submission_status_id = ?, modify_user = ?,
modify_date = ?
WHERE submission_id = ?
```

1.3.1.20 Load Submission

```
SELECT submission_id, upload_id, submission_status_id,
submission.create_user, submission.create_date,
submission.modify_user, submission.modify_date,
submission_status_lu.submission_status_id,
submission_status_lu.name,
submission_status_lu.description,
submission_status_lu.create_user,
submission_status_lu.create_date,
submission_status_lu.modify_user,
submission_status_lu.modify_date
FROM submission
WHERE submission_id = ?
```

1.3.1.20.1 Load Submissions

This is used to load multiple submissions with a single query. The only difference is that the WHERE clause becomes:

```
WHERE submission_id IN (<id_values>)
```

1.3.1.21 Load all Upload Type ids

```
SELECT upload_type_id
FROM upload_type_lu
WHERE TRUE
```

1.3.1.22 Load all Upload Status ids

```
SELECT upload_status_id
FROM upload_status_lu
WHERE TRUE
```

1.3.1.23 Load all Submission Status ids

```
SELECT submission_status_id
FROM submission_status_lu
WHERE TRUE
```

1.3.2 Sql Queries for SqlDeliverablePersistence

This section lists the SQL queries and statements that will be needed by the SqlDeliverablePersistence class, accompanied by a written explanation of what to do, when more logic than just a sequence of SQL statements is needed. For more development oriented details, see previous section.

1.3.2.1 Load deliverable with submission

```
SELECT deliverable_id, submission_id, phase_type_id,
resource_role_id, name, description, per_submission, required,
create_user, create_date, modify_user, modify_date
FROM deliverable
CROSS JOIN submission
INNER JOIN submission_status
      ON submission.submission_status_id =
            submission_status.submission_status_id
WHERE submission_status.name = 'Active' AND
deliverable.per_submission = TRUE AND
deliverable.deliverable_id = ? AND
submission.submission_id = ?
```

For selecting multiple deliverables (the loadDeliverables(long[], long[]) overload), the WHERE clause should be:

```
WHERE submission_status.name = 'Active'
      AND deliverable.per_submission = TRUE
      AND (
      (deliverable.deliverable_id = ? AND submission.submission_id = ?)
      OR (deliverable.deliverable_id = ? AND submission.submission_id = ?)
      OR (deliverable.deliverable_id = ? AND submission.submission_id = ?)
```

```
OR …)
```

1.3.2.2 Load deliverable – generic

```
SELECT deliverable_id, submission_id, phase_type_id,
resource_role_id, name, description, per_submission, required,
deliverable_lu.create_user, deliverable_lu.create_date,
deliverable_lu.modify_user, deliverable_lu.modify_date,
project_id
CROSS JOIN project
FROM deliverable_lu
WHERE deliverable_id = ?
```

For selecting multiple deliverables (the loadDeliverables(long) overload), the WHERE clause should be:

```
WHERE deliverable_id IN (?, ?, ?, …)
```

For each returned row has per_submission being true, execute the following query to select all active submission/deliverable pairs:

```
SELECT deliverable_id, submission_id
FROM deliverable
CROSS JOIN project
CROSS JOIN submission
INNER JOIN submission_status
      ON submission.submission_status_id =
            submission_status.submission_status_id
WHERE submission_status.name = 'Active'
AND deliverable.per_submission = TRUE
AND submission.project_id = project.project_id
AND deliverable.deliverable_id IN (?, ?, ?, …)
```

Put all the deliverable_id, submission_id pairs into two arrays and call the loadDeliverables(long[], long[]) overload to load all per-submission deliverables.

1.3.2.3 Load deliverables – generic

This is used to load multiple deliverables with a single query. This will result in fewer database queries being made, which will improve the performance of the application.

The query is the same as for the Load Deliverable above, but the last line should be changed to WHERE deliverable_id IN (<id_values).

1.4 Component Class Overview

UploadManager:

The UploadManager interface provides the ability to persist, retrieve and search for persisted upload and submission modeling objects. This interface provides a higher level of interaction than the UploadPersistence interface. The

methods in this interface break down into dealing with the 5 Upload/Submission modeling classes in this component, and the methods for each modeling class are fairly similar. However, searching methods are provided only for the Upload and Submission objects.

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

DeliverableManager:

The DeliverableManager interface provides the ability to persist, retrieve and search for persisted deliverable modeling objects. This interface provides a higher level of interaction than the DeliverablePersistence interface. This interface simply provides two methods to search a persistence store for Deliverables.

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

AuditedDeliverableStructure:

The AuditedDeliverableStructure 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 (as well as the id of the item) and exposes both getters and setters for these fields.

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.

NamedDeliverableStructure:

The NamedDeliverableStructure class extends the AuditedDeliverableStructure class to hold a name and description. Like AuditedDeliverableStructure, it is an abstract class. The NamedDeliverableStructure class is simply a container for the name and description. Both these data fields have the getters and setters.

This class should be very easy to implement, as all the methods just set the underlying fields.

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

Upload:

The Upload class is the one of the main modeling classes of this component. It represents an uploaded document. The Upload 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.

This class should be very easy to implement, as all the methods just set the underlying fields.

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

Submission:
The Submission class is the one of the main modeling classes of this component. It represents a submission, which consists of an upload and a status. The Submission 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.

This class should be very easy to implement, as all the methods just set the underlying fields.

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

SubmissionStatus:
The SubmissionStatus class is a support class in the modeling classes. It is used to tag a submission as having a certain status. For development, this class will be very simple to implement, as has no fields of its own and simply delegates to the constructors of the base class.

This class is mutable because its base class is mutable.

UploadStatus:
The UploadStatus class is a support class in the modeling classes. It is used to tag an upload as having a certain status. For development, this class will be very simple to implement, as has no fields of its own and simply delegates to the constructors of the base class.

This class is mutable because its base class is mutable.

UploadType:
The UploadType class is a support class in the modeling classes. It is used to tag an upload as being of a certain type. For development, this class will be very simple to implement, as has no fields of its own and simply delegates to the constructors of the base class.

This class is mutable because its base class is mutable.

Deliverable:
The Deliverable class is the one of the main modeling classes of this component. It represents an item that must be delivered for the project. The Deliverable class is simply a container for a few basic data fields, but unlike the Upload and Submission class, a deliverable is largely immutable. The data fields (except for completion date) have only get methods.

This class should be very easy to implement, as all the methods just get or set the underlying fields.

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

DeliverableChecker:

The DeliverableChecker interface is responsible for deciding if a deliverable is complete. If so, it sets the completion date of the deliverable. Only a single method exists in this interface. No concrete implementation of this interface is required in this component.

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

DeliverableFilterBuilder:

The DeliverableFilterBuilder class supports building filters for searching for Deliverables. This class consists of 2 parts. The first part consists of static Strings that name the fields that are available for searching. All DeliverableManager 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.

SubmissionFilterBuilder:

The SubmissionFilterBuilder class supports building filters for searching for Submissions. This class consists of 2 parts. The first part consists of static Strings that name the fields that are available for searching. All UploadManager 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.

UploadFilterBuilder:

The UploadFilterBuilder class supports building filters for searching for Uploads. This class consists of 2 parts. The first part consists of static Strings that name the fields that are available for searching. All UploadManager 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.

UploadPersistence:

The UploadPersistence interface defines the methods for persisting and retrieving the object model in this component. This interface handles the persistence of the upload related classes that make up the object model – Uploads, Submissions, UploadTypes, UploadStatuses, SubmissionStatuses. This interface is not responsible for searching the persistence for the various entities. This is instead handled by an UploadManager implementation.

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

SqlUploadPersistence:

The SqlUploadPersistence class implements the UploadPersistence interface, in order to persist to the database structure given in the deliverable_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 methods will need to execute several PreparedStatements in order to accomplish their tasks.

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.

DeliverablePersistence:

The DeliverablePersistence interface defines the methods for persisting and retrieving the object model in this component. This interface handles the persistence of the deliverable related classes that make up the object model – this consists only of the Deliverable class. Unlike UploadPersistence, the DeliverablePersistence interface (currently) has no support for adding items. Only retrieval is supported. This interface is not responsible for searching the persistence for the various entities. This is instead handled by a DeliverableManager implementation.

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

SqlDeliverablePersistence:

The SqlDeliverablePersistence class implements the DeliverablePersistence interface, in order to persist to the database structure given in the deliverable_management.sql script. This class does not cache a Connection to the database. Instead a new Connection is used on every method call. PreparedStatements should be used to execute the SQL statements.

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.

PersistenceUploadManager:

The PersistenceUploadManager class implements the UploadManager interface. It ties together a persistence mechanism, several Search Builder 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.

PersistenceDeliverableManager:

The PersistenceDeliverableManager class implements the DeliverableManager interface. It ties together a persistence mechanism and two Search Builder instances (for searching for various types of data). The methods in this class use a SearchBundle to execute a search and then use 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

UploadPersistenceException:

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

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

DeliverablePersistenceException:

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

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

IdAlreadySetException:

The IdAlreadySetException is used to signal that the id of one of the 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 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, “Upload removed between SearchBuilder query and loadUpload call”. This logic would not need to be in Java code, as SQL transactions can be used for this purpose.

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 1.1 or greater: Used for searching for deliverables, uploads, and submissions
- DB Connection Factory 1.0: Used in SQL persistence implementation to connect to the database. Also used by the Search Builder component.
- Custom Result Set 1.1: Returned by the Search Builder component and used to retrieve the ids of the items selected by the search.
- ID Generator 3.0: Used to create ids when new Uploads/Submissions and related objects are created.
- Configuration Manager 2.1.4: 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: Can be used to create persistence and manager implementations. There is no compile time or runtime dependency on this component. It is just a suggestion for how a client can use this component.

2.3 Third Party Components

None.

3. Installation and Configuration

3.1 Package Name

com.topcoder.management.deliverable
com.topcoder.management.deliverable.persistence
com.topcoder.management.deliverable.persistence.sql
com.topcoder.management.deliverable.search

3.2 Configuration Parameters

No direct configuration is used for this component. The Object Factory component can be used to create SqlUploadPersistences, SqlDeliverablePersistences, PersistenceUploadManagers, and PersistenceDeliverableManagers, if desired. The SearchBundles and IDGenerators needed can be configured or created programmatically. For configuration, see the component specs for these components.

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

For searching Deliverables:

```
SELECT deliverable_id
FROM deliverable
```

```
CROSS JOIN project
WHERE
```

For searching Deliverables with submission:

```
SELECT deliverable_id, submission_id
FROM deliverable
CROSS JOIN project
CROSS JOIN submission
INNER JOIN submission_status
      ON submission.submission_status_id =
            submission_status.submission_status_id
WHERE submission_status.name = 'Active' AND
deliverable.per_submission = TRUE AND
submission.project_id = project.project_id
```

For searching Submissions:

```
SELECT submission_id
FROM submission
INNER JOIN upload
      ON submission.upload_id = upload.upload_id
WHERE
```

For searching Uploads:

```
SELECT upload_id
FROM upload
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 Upload Manager*

```
SqlUploadPersistence uploadPersistence =
      new SqlUploadPersistence(
            <connection factory loaded from configuration>);

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

4.3.2 *Create an Upload and Submission (with supporting classes)*

```
// Load tagging instances (also demonstrates
// manager interactions)
UploadType uploadType = manager.getAllUploadTypes()[0];
SubmissionStatus submissionType =
      manager.getAllSubmissionStatuses()[0];
UploadStatus uploadStatus = manager.getAllUploadStatuses()[0];

// Create upload
Upload upload = new Upload(1234);
upload.setProject(24);
upload.setUploadType(uploadType);
upload.setUploadStatus(uploadStatus);
upload.setOwner(553);
upload.setParameter("The upload is somewhere");

// Create Submission
Submission submission = new Submission(823);
submission.setUpload(upload);
submission.setSubmissionStatus(submissionStatus);
```

4.3.3 *Create deliverable persistence and manager*

```
SqlDeliverablePersistence deliverablePersistence =
      new SqlDeliverablePersistence(
            <connection factory loaded from configuration>);

// The checker is used when deliverable instances
// are retrieved
DeliverableChecker checker = new <Custom deliverable checker>;

DeliverableManager manager = new PersistenceDeliverableManager(
      deliverablePersistence, checker,
      <search builders loaded from configuration: See Search
            Builder component for configuration details>,
      <id generators loaded from configuration: See Search
            Builder component for configuration details>);

// Search for deliverables (see 4.3.5)
```

4.3.4 *Save the created Upload and Submission*

```
manager.createUpload(upload, "Operator #1");
```

```
manager.createSubmission(submission, “Operator #1”);
// New instances of the tagging classes can be created through
// similar methods.

// Change a property of the Upload
upload.setProject(14424);

// And update it in the persistence
manager.updateUpload(upload, “Operator #1”);

// Remove it from the persistence
manager.removeUpload(upload, “Operator #1”);

// Submissions can be changed and removed similarly
```

4.3.5 Retrieve and search for uploads

```
// Get an upload for a given id
Upload upload2 = manager.getUpload(14402);
// The properties of the upload can then be queried
// and used by the client of this component.  Submissions
// can be retrieved similarly.

// Search for uploads
// Build the uploads – this example shows searching for
// all uploads related to a given project and having a
// given upload type
Filter projectFilter =
      UploadFilterBuilder.createProjectIdFilter(953);
Filter uploadTypeFilter =
      UploadFilterBuilder.createUploadTypeIdFilter(4);

Filter fullFilter =
      SearchBundle.createAndFilter(
            projectFilter, uploadTypeFilter);

// Search for the Uploads
Upload[] matchingUploads =
      manager.searchUploads(fullFilter);

// Submissions and Deliverables can be searched similarly by
// using the other FilterBuilder classes and the corresponding
// UploadManager or DeliverableManager methods.

// Get all the lookup table values.
UploadType[] uploadTypes = manager.getAllUploadTypes();
UploadStatus[] uploadStatuses = manager.getAllUploadStatuses();
SubmissionStatus[] submissionStatuses =
      manager.getAllSubmissionStatuses();

// Alter a lookup table entry and update the persistence
uploadTypes[0].setName(“Changed name”);
manager.updateUploadType(uploadTypes[0]);

// Lookup table entries can be created/removed through parallel
// methods to those shown in section 4.3.4
```

5. Future Enhancements

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