# Billing Cost Services 1.0 Component Specification

## 1.Design

The main goal of this project is to deliver an efficient application for automatically importing hundreds of payment/fees records from PACTS to QuickBooks application. Parts of this goal will be filtering data records (like by date/time range, by customer, by projects, etc.), viewing history of imports and audit data.

This component provides the back end billing cost services.

**Conventions:**
When referring to bean properties in this component, the standard bean dot notation is used in lieu of quoting the actually get/set methods. So instead of having

```
initiationResponse.getProblem().getId()
```

We will write

```
initiationResponse.problem.id
```

### 1.1Design Patterns

#### *1.1.1Strategy*

In the scope of the application, the provided services are used as strategies, injected into the Frontend.

#### *1.1.2DTO*

This component uses the provided entities as data transfer objects.

#### *1.1.3Dependency Injection*

Configuration takes place using dependency injection in all service and DAO classes,

## 1.2Industry Standards

Inversion of Control (IoC)
Hibernate 3.6
Spring 3.1
XML (The Spring configuration file)

## 1.3Required Algorithms

### *1.3.1Logging standard for all business methods*

All classes should have the appropriate level of logging statements.

It will log errors at Error level, potentially harmful situations at WARN level, and method entry/exit, input/output information at DEBUG level.

Specifically, logging will be performed as follows, if logging is turned on.

- Method entrance and exit will be logged with DEBUG level.
  - Entrance format: [Entering method {*className.methodName}*]
  - Exit format: [Exiting method {*className.methodName}]*. Only do this if there are no exceptions.
- Method request and response parameters will be logged with DEBUG level
  - Format for request parameters: [Input parameters[{request_parameter_name_1}:{ request_parameter_value_1}, {request_parameter_name_2}:{ request_parameter_value_2}, etc.)]]
  - Format for the response: [Output parameter {response_value}]. Only do this if there are no exceptions and the return value is not void.
  - If a request or response parameter is complex, and if it comes from this component, use its toJSONString () method. If that is not implemented (such as for list parameters then print its value using the same kind of name:value notation as above.
- All exceptions will be logged at ERROR level, and automatically log inner exceptions as well.
  - Format: Simply log the text of exception: [Error in method {*className.methodName}*: Details {*error details*}]
  - The stack trace of the error and a meaningful message.

In general, the order of the logging in a method should be as follows:

1. Method entry
2. Log method entry
3. Log method input parameters
4. If error occurs, log it and skip to step7
5. Log method exit
6. If not void, log method output value
7. Method exit

### *1.3.2Implementation of the toJSONString method*

The process of creating a JSON representation is simple, and is shown below

```
1. Create a JSON object to return: jsonObject:JSONObject
   = new JSONObject()
2. For each field:
2.1.If field is a string or date:
    jsonObject.setString(<field name>,<field value>)
2.2.If field is an int: as above but use setInt method
2.3.If field is a long or Long: as above but use setLong
    method
2.4.If field is a float: as above but use setFloat
    method
2.5.If field is a boolean: as above but use setBoolean
    method
2.6.If field is an array:
2.6.1.Create a JSON array: jsonArray:JSONArray = new
      JSONArray()
2.6.2.For each value in array, add to jsonArray, based
      on the value type, using the approach in 2.1-2.5
2.6.3.Add to object: jsonObject.setArray(<field
      name>,jsonArray)
2.7.If field is another object, then assemble it as a
    new JSONObject in its own right, using he above
    approach, then add it as above, except use
    setNestedObject method
3. return string representation: return
   jsonObject.toJSONString()
```

### *1.3.3queries*

The dashboard_billing_cost_report query should be modified to accommodate the following changes:

- The FROM clause should also retrieve the project_info_type_id and payment_details_id column values
- The WHERE clause should allow for the selection by payment_details_id OR (project_info_type_id AND contest_id)

## 1.4Component Class Overview

### *1.4.1com.topcoder.accounting.service*

**BillingCostDataService**
This interface defines the service contract for the retrieval and export of billing cost data.

**BillingCostAuditService**
This interface defines the service contract for the retrieval of billing cost export data info as well as for the creation and retrieval of account audits.

**LookupService**
This interface defines the service contract for the retrieval of all available payment areas.

### *1.4.2com.topcoder.accounting.service.impl*

**BaseService**
This is a base class for all services. It provides the common field for the Log used for all logging, and the hibernate template for all local DB interactions.

**LookupServiceImpl**
This class is an implementation of LookupService that uses Hibernate to get the PaymentArea data. Logs with the Log from the Logging Wrapper.

**BillingCostAuditServiceImpl**
This class is an implementation of BillingCostAuditService that uses Hibernate to get the billing cost audit data. Logs with the Log from the Logging Wrapper.

**BillingCostDataServiceImpl**
This class is an implementation of BillingCostDataService that uses Hibernate to get and export billing data. Logs with the Log from the Logging Wrapper.

### *1.4.3com.topcoder.accounting.entities.dao*

**IdentifiableEntity**

This is the base class for all entities that have an identification number.

**PaymentArea**

This class represents the category of payment (Studio, Software Costs etc.)

**BillingCostExport**

This class represents a record of an export of billing costs.

**BillingCostExportDetail**

This class represents the details of a record of an export of billing costs.

**AccountingAuditRecord**

This class represents a record of a single audit.

### *1.4.4com.topcoder.accounting.entities.dto*

**BillingCostReportEntry**

This class represents an entry in the billing cost report.

**BillingCostReportCriteria**

This class represents set of criteria that can be used to generate a billing cost report.

**AccountingAuditRecordCriteria**

This class represents set of criteria that can be used to generate an audit report.

**PaymentIdentifier**

This class represents a composite payment identifier.

**BillingCostExportHistoryCriteria**

This class represents set of criteria that can be used to generate a billing cost export history report.

**QuickBooksImportUpdate**

This class represents an update to the export details.

**PagedResult<T>**

This class represents a container for paged results.

### 1.5Component Exception Definitions

This component defines new exceptions.

**BillingCostServiceException**

This exception is the top-level application exception in this application. All other application exceptions in that class will extend it. It extends BaseCriticalException.

**EntityNotFoundException**

This exception is thrown in the updateBillingCostExportDetails method of BillingCostAuditService if any given billingCostExportDetailId is not found in persistence. Extends BillingCostServiceException.

**BillingCostConfigurationException**

This exception signals an issue if the configuration of any class in this application fails for any reason. It extends BaseRuntimeException.

### 1.6Thread Safety

All service-oriented classes are effectively thread-safe. None have any state. The only mutability comes from configuration via dependency injection and the Spring container does that once before usage.

Data access via HibernateTemplate is also effectively thread-safe.

## 2.Environment Requirements

### 2.1Environment

Development language: Java 1.5

Compile target: Java 1.5

### 2.2Software Components

#### *2.2.1Generic Components*

Base Exception 2.0

- Provides the base exceptions and the ExceptionData

Logging Wrapper 2.0

- Used for logging operations in all business methods.

JSON Object 1.0

- Used for serializing entity values for logging.

#### *2.2.2Custom Components*

Existing TopCoder Cockpit project

- Provides some of the existing classes

### 2.3Third Party Components

Hibernate 3.6.4

- Used for all persistence
- http://www.hibernate.org

Spring 2.5.6

- The injection container
- http://www.springsource.org/

## 3.Installation and Configuration

### 3.1Package Names

com.topcoder.accounting.entities.dao
com.topcoder.accounting.entities.dto
com.topcoder.accounting.service
com.topcoder.accounting.service.impl

### 3.2Configuration Parameters

All configuration is done using Spring dependency injection. Validation of the injections is done in the afterPropertiesSet of the InitializingBean that each class below implements. The Spring container will invoke afterPropertiesSet after injection is complete and before any business actions are invoked.

Configurable injection parameters

| parameter name | value | required |
|---|---|---|
| logger | The instance of the Log to use for logging | Yes |
| hibernateTemplate | The HibernateTemplate used for all DB interactions | Yes |

### 3.3Dependencies Configuration

#### *3.3.1Dependency configuration*

The developer should read the specifications for all components specified in section 2.2 to see how they are configured.

## 4.Usage Notes

### 4.1Required steps to test the component

Extract the component distribution.

Follow Dependencies Configuration.

Execute ‘ant test’ within the directory that the distribution was extracted to.

### 4.2Required steps to use the component

None

### 4.3Demo

#### *4.3.1Setup*

Here is a sample of the excerpt of a Spring configuration for system that would be relevant to this component

```
<bean id="logger" class="org.apache.log4j.Logger" factory-
method="getLogger">
  <constructor-arg value="myLogger"/>
</bean>
<bean id="hibernateTemplate" class="
org.springframework.orm.hibernate3.HibernateTemplate">
  <property name="sessionFactory" ref="mySessionFactory"/>
</bean>

<bean id="lookupService"
class="com.topcoder.accounting.service.impl.LookupServiceImpl">
  <property name="hibernateTemplate" ref="hibernateTemplate" />
  <property name="logger" ref="logger" />
</bean>

<bean id="billingCostAuditService"
class="com.topcoder.accounting.service.impl.BillingCostAuditSer
viceImpl">
  <property name="hibernateTemplate" ref="hibernateTemplate" />
  <property name="logger" ref="logger" />
</bean>

<bean id="billingCostDataService"
class="com.topcoder.accounting.service.impl.BillingCostDataServ
```

```
iceImpl">
  <property name="hibernateTemplate" ref="hibernateTemplate" />
  <property name="logger" ref="logger" />
  <property name="projectCategoryIds">
    <list>
      <value>1</value>
      <value>2</value>
    </list>
  </property>
  <property name="studioProjectCategoryIds">
    <list>
      <value>11</value>
      <value>12</value>
    </list>
  </property>
  <property name="statusMapping">
    <map>
      <entry><key>pending</key><value>1</value></entry>
      <entry><key>active</key><value>2</value></entry>
    </list>
  </property>
</bean>
```

*4.3.2demo*

The demo will use the services declared above. For the purpose of the demo, we assume that there are 3 payment areas:

```
// Get all payment areas
List<PaymentArea> paymentAreas =
lookupService.getPaymentAreas();
// The list would contain all payment areas, which in
our case, would be 3.

// Retrieve a billing cost report for a project for a
specific stretch of time, getting the first page of the
results
BillingCostReportCriteria billingCostReportCriteria =
new BillingCostReportCriteria();
billingCostReportCriteria.setProjectId(1L);
billingCostReportCriteria.setStartDate(// August 1,
2011);
billingCostReportCriteria.setEndDate(// August 31,
2011);
PagedResult<BillingCostReportEntry>
billingCostReportEntries =
billingCostDataService.getBillingCostReport(criteria,1,1
0);
// This result would get the first 10 entries in the
report for a specific project in the month of august, as
part of a monthly billing report

// Export the above-retrieved entries
List<PaymentIdentifier> paymentIds = // payment IDs
assembled from the above entries
TCSubject user = // the current user;
long paymentAreaId = 1L; // assume this is the area of
payment we want, such as studio
billingCostDataService.exportBillingCostData(paymentIds,
1,user);
// This action would result in the export of these
entries, as identified by their PaymentIdentifiers
```

The following methods can be use to get export data, get an identifier, and to manage audits.

```
// Get all exports for a given payment area for a
spedific date range, showing the first page
BillingCostExportHistoryCriteria
billingCostExportHistoryCriteria = new
BillingCostExportHistoryCriteria();
billingCostExportHistoryCriteria.setPaymentAreaId(1L);
billingCostExportHistoryCriteria.setStartDate(// August
1, 2011);
billingCostExportHistoryCriteria.setEndDate(// August
31, 2011);
PagedResult<BillingCostExport> exports =
billingCostAuditService.getBillingCostExportHistory(bill
ingCostExportHistoryCriteria,1,10);
// This result would get the first 10 entries in the
report for a specific paymet area in the month of
august, as part of a monthly export report. Note that
these results may include the result we exported above.

// Gets all details for a single cost export, showing
the first page
long billingCostExportId = // one of the above exports
PagedResult<BillingCostExportDetail> details =
billingCostAuditService.getBillingCostExportDetails(bill
ingCostExportId,1,10);
```

```
// Another way of searching details is to search for all
that are now in quickbooks, showing the first page
PagedResult<BillingCostExportDetail> details2 =
billingCostAuditService.getBillingCostExportDetails(true
,1,10);

// This method creates a new audit record
AccountingAuditRecord accountingAuditRecord = // new
audit record with data
billingCostAuditService.auditAccountingAction(accounting
AuditRecord);

// Gets all audits records for a given action for a
spedific date range, showing the first page
AccountingAuditRecordCriteria
accountingAuditRecordCriteria = new
AccountingAuditRecordCriteria();
accountingAuditRecordCriteria.setAction("updateBillingCo
stExportDetails");
accountingAuditRecordCriteria.setStartDate(// August 1,
2011);
accountingAuditRecordCriteria.setEndDate(// August 31,
2011);
PagedResult<AccountingAuditRecord> auditRecords =
billingCostAuditService.getAccoutingAuditHistory(account
ingAuditRecordCriteria,1,10);

// Perfrom some updates
List<QuickBooksImportUpdate> updates = // updates
billingCostAuditService.updateBillingCostExportDetails(u
pdates);

// Get the latest invoice number
String latestInvoiceNumber =
billingCostAuditService.getLatestInvoiceNumber();
```

## 5.Future Enhancements

None