Time Entry 1.1 Component Specification

1. Design

The Time Entry component is part of the Time Tracker application. It provides an abstraction of a time log entry that an employee enters into the system on a regular basis. This component handles the logic of persistence for this.

Note that additions to this document have been noted in red and changes in blue.

The 1.1 version introduces a number of improvements and additions:

- Package name has changed and is now com.cronos.timetracker.entry.time instead of the previously used com.topcoder.timetracker.entry.time.
- RejectReason has been added to the design so that time entries can be rejected with a proper rejection reason provided.
- Search capabilities over time entry data (which includes linked reject reason data) have been added that allow users to search for specific time entries using a wide variety of search criteria.

Version 1.1 will enhance the existing version 1.0 design and maintain the same underlying API. New classes and methods will be added to meet the additional requirements.

Each table in a data store is represented by a DataObject class, with each instance representing a row in that table. A corresponding DAO (Data Access Object) class exists to handle CRUD (Create, Read, Update, and Delete) functionality for the DataObject. As such, there exists a 1-1 mapping between DataObjects and DAOs. Thus, there is a TimeEntry entity backed by a TimeEntryDAO, a TaskType entity backed by a TaskTypeDAO, and a TimeStatus entity backed by TimeStatusDAO.

There are two means of returning information. The first one is to simply return the record as is, with actual foreign Ids, which the application can chose to query using other DAOs. The second one is for the data object to recursively contain references to parent data objects. The framework allows both because a designer controls how a DAO implementation behaves. This design will use the first means, and all three data objects will return the foreign ids where applicable.

The 5 CRUD operations provided by the DAO interface are independent of the persistence mechanism that an implemention can use. Because this component is initially geared for JDBC connectivity with a Database, the DAO implementations will work with JDBC connections. Thus, each CRUD method is responsible for obtaining and closing JDBC resources (connections, statements, result sets). The DBConnection Factory component is used to obtain connections. To that end, each DAO implementation has a constructor that accepts a connection name (also obtained via the Config Manager in the process of

matching a DataObject to a DAO) that it uses to obtain a Connection from the DBConneciton Factory.

In order to support local transactions, the DAO interface contains three methods that allow the application to manage the JDBC connection for the DAO. JDBC DAOs in this application implement these methods. In this scenario, the DAO uses a connection provided by the application, not from the DBConnection Factory, and does not close the Connection once it is done. This allows the application to call several DAO methods on one Connection and commit it when it is done (done by setting the autoCommit() to false, and then calling commit(), when done).

Because there is much common functionality associated with each database operation, a base DAO class implements most of the common CRUD functionality, and leaves several methods for the DAO implementation to provide the specificity. For example, the delete() method performs basically the same task regardless of which table is queried. The only difference is the name of the table. Thus, an abstract method – getDeleteSqlString – is defined and implemented by each concrete DAO class.

Some such helper methods are already implemented but can still be overridden. During the create() method, a primary Id is generated by the TC component IDGenerator in the helper method *setDataObjectPrimaryId()*. All three concrete implementations use this facility, but future implementations can override it.

Anatomy of the proposed enhancements

The 1.1 enhancements revolve around three aspects:

- **Batch operations**
  The design will deal with this on two levels. Since batch operations have the convenience of stacking many operations together, this design will provide both a *synchronous* mode (i.e. caller will wait for the batch process to end) as well as an *asynchronous* mode, where the caller can assign a batch job and then be notified when it is done.
- **Reject reason capabilities**
  This is pretty straightforward application of requirements. New tables will be created and new API will be incorporated.
- **Search operations**
  The design will create a search framework where individual search criteria such as a date or description can be combined together using more sophisticated boolean operators (AND, OR, NOT) as well as range (from, to, in) and approximation (LIKE) operators to create complex searches.

Batch operations

This design will define the batch operations in terms of the primitive CRUD operations that are already available through the TimeEntryDAO.

Important aspects to remember is that the 1.1 version of the DAO will actively also deal with linking and unlinking of RejectReason data for a particular TimeEntry. This means that batch operations will one by one make calls to the CRUD operations of the TimeEntryDAO which will also deal with linking and unlinking of RejectReason dependencies.

The Batch API will provide a mechanism for execution details for each statement that was part of the sent batch: it will return an array of ints, which will provide completion or error information for each primitive statement executed.

The synchronous mechanism is very straightforward since all our API will do is simply wrap around the TimeEntryDAO CRUD. To simplify the interaction of the API we will allow for a ResultHandler to be registered with each batched call which will allow the user (if they so prefer) to register a handler to get all the information about the batch run. This way we can pass back to the user, a detailed report on the run. The asynchronous mechanism will work in a very similar manner. A ResultHandler will be registered and result data will be Pushed to it when the batch run is completed. Of course the asynchronous mode will be thread driven.

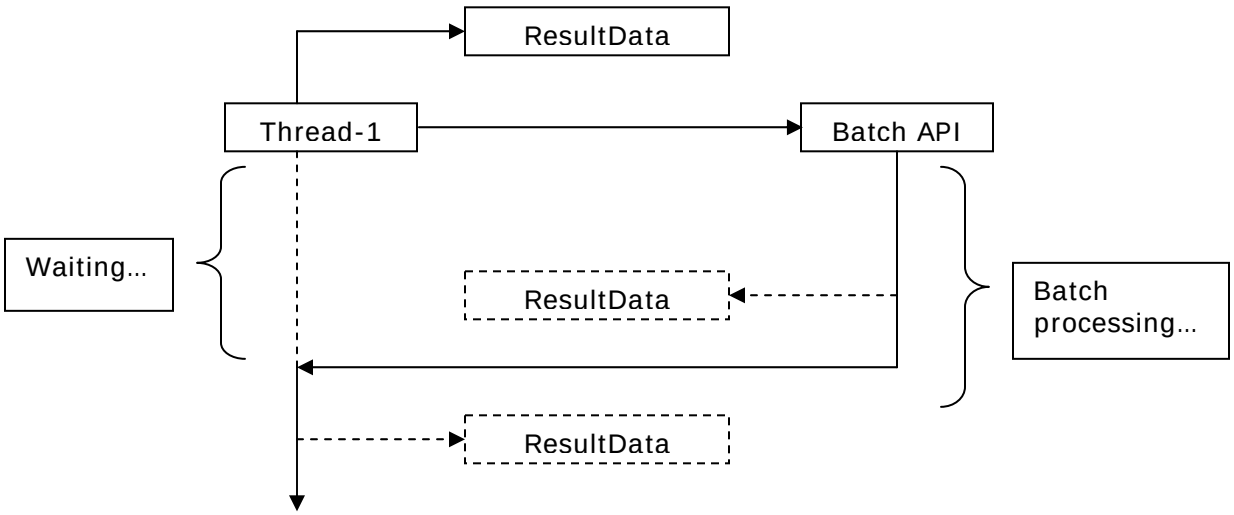
Incorporating batch operation into existing API
Current Time-Entry API presents some integration challenges since there are a number of assumptions made by previous designer that will have to be honored as they act almost like a contract. These are:

> There is a generic, super DAO that represents all CRUD actions in a generic, Object type driven API. Data is based on a generic DataObject class and thus all results have to be cast to their proper type by the user.
>
> The design allows for the user to provide the API with an external instance of a Connection object.
>
> All DAOs are created through a DAO factory and thus the user will have to be aware of proper casting to be done.

This design will try to fit in. What we will do is we will create two new interfaces: a `BatchDAO` and an `AsynchBatchDAO`, which will then be implemented by the `BaseDAO` abstract class. The BaseDAO abstract class will actually implement all the batch methods in an *Operation Not Supported* mode. This is done to accommodate those DAOs that do not need to have batch operations defined but at the same time do not want to go through the process of setting each not supported method to throw the `UnsupportedOperationException`.

Retrieving results when running a batch operation
The basic operation of result retrieval is quite simple. In either mode we simply register a ResultData data receptacle and then look up the results through this receptacle.

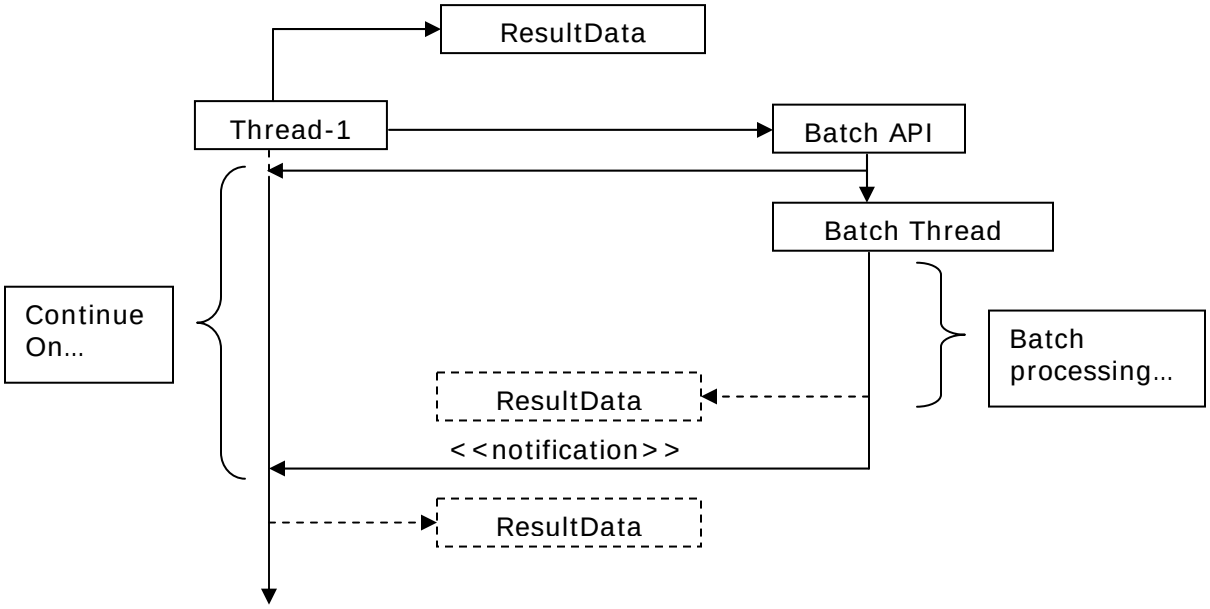
Synchronous mode



The above diagram can be described as follows:

- The main thread will create a ResultHandler instance and register it prior to the batch call.
- It will then execute a synchronous (i.e. blocking) call to the Batch API
- Batch API will process the statements while the thread blocks and waits.
- When Batch API is done it will put the results into the registered ResultHandler and will then return control to the thread.
- The thread can inspect the results of the batch run if it wants to.

Asynchronous mode



The above diagram can be described as follows:

- The main thread will create a ResultHandler instance and register it prior to the asynch batch call.
- It will then execute a synchronous (i.e. blocking) call to the Batch API, which will create a sub-thread and will return control right away to the caller thread, which can then continue doing other things.
- Batch API will process the statements while the main thread does its own independent work.
- When Batch API is done it will put the results into the registered ResultHandler and will then notify the main thread that results are available. To do this the main thread will have to implement a specific notification interface, which will be passed in through with the ResultHandler

Handling Batch exceptions coming from wrapped DAO

Exceptions will be handled by storing for each failed operation, the very exception that was, in an array of exceptions. This array will simply mirror in a 1-1 manner the batch operations themselves in a manner that if we wanted to see if the batch operation at location (i.e. index) 5 failed we simply check if we have an exception in the exceptions array at location 5. If we have a null then the operation was successful, and if we have a non-null reference then we have the instance of the exception that was caught during operation at location 5. It is recommended that we set autocommit to false when using batch updates "for proper error handling." Doing so also allows all the benefits of transaction processing.

This is important since Time Entry 1.0 makes an architectural decision to allow the users to introduce their own Connection object. If the object is supplied from outside then we do not follow the above directive. But of the connection is created internally then the developer should follow the given format of exception handling:

```
try
{
    con.setAutoCommit( false );
    ...
    bError = false;

    // execute the statements
    for(int i=0; i< dataObjects.length; i++){
        try{
              // execute the specific DAO CRUD action for the element
              //
        }catch{
              // any caught exception should be added to the exception
              // array at index i
        }
   } // end for

}

// catch blocks
...

finally
{
    // determine operation result
    for (int i = 0; i < exceptionList.length; i++)
    {
      iProcessed = aiupdateCounts[i];
      if( iProcessed != null)
      {
        // statement was successful
        ...
      }
      else
      {
        // error on statement
        bError = true;
        break;
      }
    } // end for

    if( bError )
    {
      con.rollback();
    }
    else
    {
      con.commit();
    }
} // end finally
```

Reject reason capabilities

Implementation of this will be very simple. In the design we add a new DAO implementation for this (which will do nothing to the batch operations since it

makes no sense to support those for such a DAO). We also have to create a value object for the reject reason entity.

Search options

Here we will try to formalize the aspect of searching into an extensible mini-framework.

Basic Search Criteria

Searching will be done through the notion of criteria. Consider searching for time entries that were created by a specific user. How could we formalize this? This is quite simple since we are dealing with fields really. The example just mentioned would be simply "Find all time entries where CreationUser = user" Thus what we need is the ability to match any field in TimeEntries entity with an actual value or value combinations. Thus the criteria would be the field names. This can be abstracted into the notion of TimeEntryCriteria.

Thus what we will do is we will abstract all the Time entry fields so that the user can use them in a search. Thus we will have the following mapping to requirements:

Requirement	Criteria	Comments
1.2.3.1	TimeEntryCritera.DESCRIPTION	String type
1.2.3.2	TimeEntryCritera.TIME_STATUS_ID	Long type
1.2.3.3	TimeEntryCritera.TASK_TYPE_ID	Long type
1.2.3.4	TimeEntryCritera.CREATION_USER TimeEntryCritera.MODIFICATION_USER	String type
1.2.3.5	TimeEntryCritera.BILLABLE_FLAG	Expecting String with values "false" or "true"
1.2.3.6	TimeEntryCritera.REJECT_REASON_ID	Long type
1.2.3.7	TimeEntryCritera.HOURS TimeEntryCritera.DATE	Long type String date in the format of the database
1.2.3.8	TimeEntryCritera.CREATION_DATE TimeEntryCritera.MODIFICATION_DATE	String date in the format of the database

The above will be used as entries into a search expression.

Search Expressions

Search expressions are what we will use to combine criteria to actually create searches that can be executed. We will have the following possible expressions:

Complex expressions

Complex expressions are expressions that can aggregate other expressions. Boolean expressions including OR, AND, and NOT

Simple expressions
Simple expressions are the basic building blocks of expressions.
Comparison expressions which will allow for equality testing such as ==, !=, >=, <=, etc… of criteria a values.
Substring expressions, which will allow for any string matching including sub string matching.
Range expressions, which will allow for such searches that would include a date range or a time range for example. These are always inclusive. Of course the user can build range expressions by using Comparison and Boolean expressions.

Evaluation and validation of expressions
Once we have an expression how do we evaluate it? How do we validate it?

Evaluation of an expression
All of the mentioned expressions are supported by SQL and thus we can simply have an evaluator that constructs the appropriate SQL string.
Thus Boolean-expressions using AND, OR, and NOT are very easily mapped into SQL's boolean operators. Comparison-Expressions are likewise mapped almost 1 to 1 with SQL operators. Substring-Expressions will utilize the LIKE operator (using the form of %substring%) Finally, Range-Expressions will be easily mapped using a combination of Boolean and comparison operators.

Validation of expressions
We will delegate the actual validation of the query to the database. Simply of the query makes no sense then the actual search will generate a TimeEntrySearchException.

1.1 Design Patterns

Factory:
Used in the DAOFactory class.

Template Method:
Used in BaseDAO class. All CRUD methods rely on helper methods, whose functionality is provided by subclasses.

Strategy:
Also used in the DAOFactory – A user can request a DAO and not have to worry about the specifics of how the DAO, or even which DAO, performs the persistence action.

DAO:
Used by DAO classes to provide data access services through a common API.

Value Object:
Used by DataObject and its subclasses.

Decorator Pattern
Used to decorate the batch operations with the ability to create asynchronous batching by wrapping around the synchronous batch operations implementation.

Listener Pattern
Utilized for asynchronous notification of the caller when an asynchronous batch operation is complete.

1.2 Industry Standards

SQL, JDBC

1.3 Required Algorithms

There are not complicated algorithms in this design, and most methods will contain implementation notes. This section will reiterate more salient algorithm information. All of this information will be found as implementation notes in the appropriate places in Poseidon tabs. Some sequence diagrams also illustrate most of these steps.

The first 5 sections show the CRUD method SQL statement strings, and how they are filled and processed. Since most CRUD operations are performed using JDBC PreparedStatements, the filling information will show how to match DataObject members to statement parameters or how to write ResultSet parameters into the DataObject, where applicable.

The subsequent sections deal with other matters, such as reading of property files for DAO instantiation in the factory, id generation, etc.

1.3.1 DAO create() method.

The SQL statements follows the following paradigm:

```
INSERT INTO table_name (column1, column2,…) VALUES (?,?…)
```

The actual SQL statements are:

TimeEntryDAO:

```
INSERT INTO TimeEntries(TimeEntriesID, TaskTypesID,
TimeStatusesID, Description, EntryDate, Hours, Billable,
CreationUser, CreationDate, ModificationUser, ModificationDate
VALUES (?,?,?,?,?,?,?,?,?,?,?)
```

TaskTypeDAO:

```
INSERT INTO TaskTypes(TaskTypesID, Description, CreationUser,
CreationDate, ModificationUser, ModificationDate VALUES
(?,?,?,?,?,?)
```

TimeStatusDAO:

```
INSERT INTO TimeStatuses(TimeStatusesID, Description,
CreationUser, CreationDate, ModificationUser, ModificationDate
VALUES (?,?,?,?,?,?)
```

When filling the PreparedStatement in the fillCreatePreparedStatement() method, the following PreparedStatement setter and value is to be used. The variable "user" is the one passed with the method. "date" is instantiated internally to the current date. Note that the date must be converted to a java.sql.Date in order to be set.

TimeEntryDAO:
setInt(1, TimeEntry.primaryId)
setInt (2, TimeEntry.taskTypeId)
setInt (3, TimeEntry.timeStatusId)
setString(4, TimeEntry.description)
setDate(5, TimeEntry.date)
setFloat(6, TimeEntry.hours)
setBoolean(7, TimeEntry.billable)
setString(8, user)
setDate (9, date)
setString (10, user)
setDate (11, date)

TaskTypeDAO:
setInt (1, TaskType.primaryId)
setString(2, TaskType.description)
setString(3, user)
setDate (4, date)
setString (5, user)
setDate (6, date)

TimeStatusDAO:
setInt (1, TimeStatus.primaryId)
setString(2, TimeStatus.description)
setString(3, user)
setDate (4, date)
setString (5, user)
setDate (6, date)

1.3.2 DAO update() method.

The SQL statements follows the following paradigm:
```
UPDATE table_name SET column1=?,column2=? WHERE primary_ID=?
```

The actual SQL statements are:
TimeEntryDAO:
```
UPDATE TimeEntries SET TaskTypesID =?, TimeStatusesID =?,
Description=?, EntryDate =?, Hours=?, Billable=?,
ModificationUser=?, ModificationDate=? WHERE TimeEntriesID=?
```

TaskTypeDAO:
```
UPDATE TaskTypes SET Description=?, ModificationUser=?,
ModificationDate=? WHERE TaskTypesID=?
```

TimeStatusDAO:
```
UPDATE TimeStatuses SET Description=?, ModificationUser=?,
ModificationDate=? WHERE TimeStatusesID=?
```

When filling the PreparedStatement in the fillUpdatePreparedStatement() method, the following PreparedStatement setter and value is to be used. The variable "user" is the one passed with the method. "date" is instantiated internally to the current date. Note that the date must be converted to a java.sql.Date in order to be set.

TimeEntryDAO:
setInt(1, TimeEntry.taskTypeId)
setInt (2, TimeEntry.timeStatusId)
setString(3, TimeEntry.description)
setDate(4, TimeEntry.date)
setFloat(5, TimeEntry.hours)
setBoolean(6, TimeEntry.billable)
setString(7, user)
setDate (8, date)
setInt (9, TimeEntry.primaryId)

TaskTypeDAO:
setString(1, TaskType.description)
setString(2, user)
setDate (3, date)
setInt (4, TaskType.primaryId)

TimeStatusDAO:
setString(1, TimeStatus.description)
setString(2, user)
setDate (3, date)
setInt (6, TimeStatus.primaryId)

1.3.3 DAO get() method.

The SQL statements follows the following paradigm:
```
SELECT * FROM table_name WHERE primary_ID=?
```

The actual SQL statements are:
TimeEntryDAO:
```
SELECT * FROM TimeEntries WHERE TimeEntriesID=?
```
TaskTypeDAO:
```
SELECT * FROM TaskTypes WHERE TaskTypesID=?
```
TimeStatusDAO:
```
SELECT * FROM TimeStatuses WHERE TimeStatusesID=?
```

When retrieving values from the ResultSet, the following getters should be used to fill the specified members.

TimeEntryDAO:

getInt (“TimeEntriesID”)	→ TimeEntry.primaryId
getInt (“TaskTypesID”)	→ TimeEntry.taskTypeId
getInt (“TimeStatusesID”)	→ TimeEntry.timeStatusId
getString(“Description”)	→ TimeEntry.description
getDate(“EntryDate”)	→ TimeEntry.date
getFloat(“Hours”)	→ TimeEntry.hours
getBoolean(“Billable”)	→ TimeEntry.billable
getString(“CreationUser”)	→ TimeEntry.creationUser
getDate (“CreationDate”)	→ TimeEntry.creationDate
getString (“ModificationUser”)	→ TimeEntry.modificationUser
getDate (“ModificationDate”)	→ TimeEntry.modificationDate

TaskTypeDAO:

getInt (“TaskTypesID”)	→ TaskType.primaryId
getString(“Description”)	→ TaskType.description
getString(“CreationUser”)	→ TaskType.creationUser
getDate (“CreationDate”)	→ TaskType.creationDate
getString (“ModificationUser”)	→ TaskType.modificationUser
getDate (“ModificationDate”)	→ TaskType.modificationDate

TimeStatusDAO:

getInt (“TimeStatusesID”)	→ TimeStatus.primaryId
getString(“Description”)	→ TimeStatus.description
getString(“CreationUser”)	→ TimeStatus.creationUser
getDate (“CreationDate”)	→ TimeStatus.creationDate
getString (“ModificationUser”)	→ TimeStatus.modificationUser
getDate (“ModificationDate”)	→ TimeStatus.modificationDate

1.3.4 DAO getList() method.

The SQL statements follows the following paradigm:

```
SELECT * FROM table_name [WHERE user-provided_where_clause]
```

The user will pass a complete where clause (without the WHERE keyword), and the method will append it to the statement. The method could verify that the WHERE keyword is not present in the passed whereClause, or simply try to execute the statement. If the user does not pass a where clause, then the WHERE keyword is not added to the SQL statement and the method gets all records.

The actual SQL statements are:

TimeEntryDAO:

```
SELECT * FROM TimeEntries [WHERE …]
```

TaskTypeDAO:

```
SELECT * FROM TaskTypes [WHERE …]
```

TimeStatusDAO:

```
SELECT * FROM TimeStatuses [WHERE …]
```

When retrieving values from the ResultSet, the following getters should be used to fill the specified members.

TimeEntryDAO:

getInt (“TimeEntriesID”)	→ TimeEntry.primaryId
getInt(“TaskTypesID”)	→ TimeEntry.taskTypeId
getInt (“TimeStatusesID”)	→ TimeEntry.timeStatusId
getString(“Description”)	→ TimeEntry.description
getDate(“EntryDate”)	→ TimeEntry.date
getFloat(“Hours”)	→ TimeEntry.hours
getBoolean(“Billable”)	→ TimeEntry.billable
getString(“CreationUser”)	→ TimeEntry.creationUser
getDate (“CreationDate”)	→ TimeEntry.creationDate
getString (“ModificationUser”)	→ TimeEntry.modificationUser
getDate (“ModificationDate”)	→ TimeEntry.modificationDate

TaskTypeDAO:

getInt (“TaskTypesID”)	→ TaskType.primaryId
getString(“Description”)	→ TaskType.description
getString(“CreationUser”)	→ TaskType.creationUser
getDate (“CreationDate”)	→ TaskType.creationDate
getString (“ModificationUser”)	→ TaskType.modificationUser
getDate (“ModificationDate”)	→ TaskType.modificationDate

TimeStatusDAO:

getInt (“TimeStatusesID”)	→ TimeStatus.primaryId
getString(“Description”)	→ TimeStatus.description
getString(“CreationUser”)	→ TimeStatus.creationUser
getDate (“CreationDate”)	→ TimeStatus.creationDate
getString (“ModificationUser”)	→ TimeStatus.modificationUser
getDate (“ModificationDate”)	→ TimeStatus.modificationDate

1.3.5 DAO delete() method.

The SQL statements follows the following paradigm:

```
DELETE FROM table_name WHERE primary_ID=?
```

The actual SQL statements are:

TimeEntryDAO:

```
DELETE FROM TimeEntries WHERE TimeEntriesID=?
```

TaskTypeDAO:

```
DELETE FROM TaskTypes WHERE TaskTypesID=?
```

TimeStatusDAO:

```
DELETE FROM TimeStatuses WHERE TimeStatusesID=?
```

1.3.6 Matching DataObjects to DAOs in the Factory

The factory will query the Config Manager component for the target name of the DAO to instantiate and the name of the connection to be used when creating a

connection against the DBConnectionFactory. This name, as well as the provided namespace will be passed in the DAO constructor. Construction is done via reflection.

1.3.7 Generating primary keys

The IDGenerator will be used to obtain the next key. Please refer to IDGenerator's documentation for more details on how this is done.
The actual key used to request the key for the right table will the fully qualified class name of the requesting DataObject implementation.

1.3.8 Requesting Connections from DBConnection Factory

The DBConnectionFactory will be used to obtain the connection. Please refer to DBConnectionFactory's documentation for more details on how this is done. This component will use Informix as the database server and the connection will be created from there via jdbc.

1.3.9 Local Connections vs Factory Connections

Each CRUD operation in a DAO obtains a Connection for the duration of the operation. This is done by requesting a Connection from the DBConnection Factory. The operation then closes this Connection once it is done. This is the usual and default behavior. But, if an application wishes to perform local transactions, it can override this mechanism and pass its own Connection object, which each operation must use (and not close) for as long as the application wishes it.

Thus, each DAO in this component must be able to function in one of two modes. It does this in the following manner.

Each CRUD operation in the BaseDAO begins with a call to obtain a Connection and finishes with a call to close resources (the Connection, Statement, and ResultSet, if applicable). The two methods – *createConnection()* and *closeResources(),* provide separation of concerns for each CRUD operation, and it no longer must concern it self with their specifics.

When each DAO is constructed, it receives the name of the Connection and a namespace it is to use with the DBConnection Factory. BaseDAO defines a flag – *useOwnConnection* – which signals whether the DAO is to get its own connection via the DBConnectionFactory (true), or use the Connection stored in its connection member (false). Similarly, when closing resources, if the flag is on, then the Connection must be closes, otherwise it is not to be closed. The default value of this flag is true. The pseudocode below will clarify further.

createConnection()

```
if (useOwnConnection)
     obtain Connection from DBConnection Factory
else
     use Connection stored in the connection member.
```

closeResources()

```
if (useOwnConnection)
     Connection.close()
else
     Do nothing to Connection.
```

The component defines three methods in the DAO interface to facilitate direct management of Connections by the application – *setConnection()*, *getConnection()*, and *removeConnection()*. Through the *setConnection()* method, the application passes a Connection object to the DAO, which the DAO will use in all its operations from henceforth. Specifically, this method sets the *useOwnConnection* flag to off. Once the application is done with its local transaction, it calls the *removeConnection()* method, which nullifies the connection member in the DAO, and sets the *useOwnConnection* flag to on again, so the DAO returns to its normal mode of managing Connections. It can't be stressed enough how tightly coupled the *setConnection()* and *removeConnection()* methods are. The code fragments below will clarify further.

setConnection(Connection connection)

```
     this.connection = connection;
     useOwnConnection = false;
```

removeConnection()

```
     this.connection = null
     useOwnConnection = true;
```

The getConnection() simply retrieves the connection member value.

1.3.10 Searching and evaluating

Searching and evaluating for SQL based searches is very simple. All we need to do is have the following:

- Each boolean search expression is simply expanded into an SQL syntax string by taking its operator method (and, or, not) and producing the equivalent SQL AND, OR, and NOT syntax. This is done by taking each one of the sub expressions and expanding it in turn recursively. It is also important to note that we should put each such expression into parenthesis so that we have no precedence ambiguities. Thus lets say that the user creates the following:

  expression.and(a, b);

  We would expand the expression into SQL as follows:

  (a AND b)

  and then a and b would be expanded in turn as well.
- Each comparison expression is also expanded into a simple SQL syntax string. But comparison is not recursive (it is a leaf-node) in an expression tree. For example lets say that the user has the following:

expression.equals(a, 2)

We would expand this into

field.name = 2

where field.name is how the criteria would be mapped.

Follow the above for the other types of expressions.

1.4 Component Class Overview

DataObject:

Abstract class that provides the common primaryId member with get/set accessor methods. All entities will extend this class and define additional members as needed.

BaseDataObject:

Abstract class. Extends DataObject to provide common Time Entry fields and get/set accessor methods.

DAO:

Interface. Defines five CRUD(Create, Read, Update, and Delete) database related operations for the DataObject. It also defines other three methods that can be used to external manage connection for the DAO if local transactions are desired. For non-JDBC DAOs, it is recommended that these three methods throw UnsupportedOperationException.

BaseDAO:

BaseDAO class implements the DAO interface. It contains a connection name and connection namespace to get connection from DBConnection factory in normal connection-obtaining schema, and it also allows the running application to control the connection and exercise local transaction control. It implements the five CRUD(Create, Read, Update, and Delete) database related operations. These methods rely on helper methods to provide entity-specific information such as Connection creation, resource closing, primarykey generation and etc. Some of the helper methods are implemented here for its common sense.

DAOFactory:

Defines one method that instantiates the DAO that a DataObject implementation will use to persist itself.

TimeEntry:

This class represents the TimeEntry entity. Extends BaseDataObject and uses its primaryId member for holding the TimeEntriesID defined in the database.

TimeEntryDAO:

This class extends BaseDAO and implements all abstract methods to provide TimeEntry-specific information.

TaskType:

This class represents the TaskType entity. Extends BaseDataObject and uses its primaryId member for holding the TaskTypesID defined in the database.

TaskTypeDAO:
This class extends BaseDAO and implements all abstract methods to provide TaskType-specific information.

TimeStatus:
This class represents the TimeStatus entity. Extends BaseDataObject and uses its primaryId member for holding the TimeStatusID defined in the database.

TimeStatusDAO:
This class extends BaseDAO and implements all abstract methods to provide TimeStatus-specific information.

BatchDAO <<interface>>
This is a contract for batch operations. This assumes synchronous operation for all methods. The batch contract assumes that each batch operation can be set as atomic (i.e. all or nothing) or non-atomic (which means that the we treat each operation in the batch as independent)
The batch CRUD provided comes with the ability to set a result handler which can be used by the caller to have the batch operation API deposit the results individually for each batched operation. User code will be able to get each and every operation and its status. Currently the following implementations of this class must support the following result behavior: It must return an array of BatchOperationException instances if any exceptions were caught during the operations.

AsynchBatchDAOWrapper <<concrete>>
This is a wrapper (or close to a wrapper) around any object that implements the BatchDAO interface. This will simply delegate all the calls to the underlying batch DAO but in an asynchronous manner. The user will have to provide a valid implementation of a ResultListener interface to be notified when the batch is done. This is an idea solution for larger batches that might take considerable time and thus would benefit from being run in the background.

ResultData <<concrete>>
This is a simple class, which allows the caller to retrieve result exceptions, operation result information, as well as the actual result of the operation itself (this is only true when dealing with batched reads)

ResultListener <<interface>>
This is a simple notification interface, which would be implemented by the caller of the asynchronous batch process. When the process is done it will notify the caller via this interface and will deposit all the necessary result information as part of the notification (using ResultData instance)

RejectReasonDAO <<concrete>>
This is an implementation of the BaseDAO abstract class to provide data access functionality for the reject Reason requirements. This DAO will not have batch operations capabilities.

RejectReason <<concrete>>
This implementation of the BaseDataObject abstract class will provide a value object for the RejectReasonDAO class. This class represents a single reject reason record in the database.

TimeEntryCriteria<<concrete class>>
This is a simple listing of available search criteria elements for Time Entries. This is an enum, which basically gives the user an enumeration of all, currently available, criteria that can be used in searches.

SearchExpression <<interface>>
This is a simple interface, which tags an implementing class as being a search expression. Will have a simple getExpressionString() method.

BooleanExpression <<concrete>>
This is a representation of a boolean expression which simply, through and(), or() and, not() methods allows to combine other search expressions. A typical usage would be:
BooleanExpression expression = BooleanExpression.and(expressionA, expressionB);

ComparisonExpression <<concrete>>
This is a representation of a comparison expression, which simply, through equals(), notEquals(), lessThanOrEqual(), greaterThan(), and greaterThanOrEqual() methods allows to combine search criteria and values. A typical usage would be:
ComparisonExpression expression = ComparisonExpression.equals(TimeEntryCriteria.CREATION_USER, “Ivern”).

SubstringExpression <<concrete>>
This is a representation of a Substring expression, which simply, through a single contains() method, allows us to specify a search criteria and a partial value to be matched in that field as a substring. A typical usage would be:
SubstringExpression expression = SubstringExpression.contains(TimeEntryCriteria.DESCRIPTION, “free”);

RangeExpression <<concrete>>

This is a representation of a range expression, which simply, through from(), fromTo(), and to() methods , allows us to specify a search criteria and one of two range values. A typical usage would be
RangeExpression expression =
RangeExpression.fromTo(TimeEntryCriteria.HOURS, 10, 25);

ExpressionEvaluator <<interface>>
This is a general contract for running search expressions. It simply takes a search expression, evaluates it and returns the search results (which is an array of Objects)

SQLBasedTimeEntryCriteriaExpressionEvaluator <<concrete>>
This is a simple, SQL based evaluator which takes in a SearchExpression, converts it into some SQL statements, and submits it to the data base and returns (hopefully) an array of TimeEntry Objects that have met the input criteria.
If the criteria were invalid or the search expression evaluation has failed then a TimeEntrySearchException will be thrown.

1.5 Component Exception Definitions

This component defines two custom exceptions that use exception chaining. As such, they act as wrappers for exceptions encountered inside the methods. These internal exceptions are to be wrapped in these custom exceptions.

DAOFactoryException:
Thrown if anything goes wrong in the getDAO method in the DAOFactory. Currently, some causes would be if the passed object is not a DataObject, if there is an exception thrown by the ConfigManager while obtaining configuration info, or when using reflection to construct the DAO.

DAOActionException:
Thrown if anything goes wrong in any DAO methods and all helpers except closeResources, which throws SQLException.

The component also throws predefined exceptions:

NullPointerException:
For 1.1 enhancements we do not use this exception at all.
All Object parameters in the DAO classes must not be null, and all methods with these parameters will throw this exception if they are. The only exception, no pun intended, is the getList method, where a null whereClause implies no constraints. DataObjects do not check for any null data.

IllegalArgumentException:

User, namespace, nameConn must not be empty in the DAO implementations. DataObjects do not check for any empty data.
For 1.1. enhancements This will be thrown in situations where the input is considered illegal (this includes illegal null pointer input). Some examples follow: Empty String (i.e. a string which after it is trimmed has a length of 0)

SQLException:
Thrown by BaseDAO.closeResources() if there is a problem while closing these resources.

SearchException
This is an exception that signals that there was an issue with a search operation.

BatchOperationException
This exception signals that something went wrong with the batch operation.

1.6 Thread Safety

The component does not handle thread safety in its domain, and in most scenarios, thread safety will not be a concern. Most scenarios call for use of global transaction control, and DAOs will obtain their connections per call from the DBConnection Factory, which is thread-safe, and the DAOs will not hold any state. Also, the IDGenerator is thread-safe. Thus, even if the DAOs are cached, operations will be effectively atomic.

In these circumstances, the mutability of the DataObjects might pose some problems. For instance, just after a DAO verifies that DataObject members are valid, but before the values are sent to the data base, a thread with access to this data object could invalidate the member values. This would result in a SQLException and no damage would be done. But overall, most scenarios call for a DataObject to be associated with a request, which will usually be associated with just one thread.

There is one scenario that could be potentially pernicious: local transactions. In this scenario, many DAOs could be sharing a Connection. If two threads attempt to perform overlapping transactions on the same Connection, the transactions will not be isolated. Thus it is imperative that an application running this scenario provide thread-safety on its own and allow only one transaction on a Connection at a time.

1.1. Enhancements follow the thread-safety standard set out by version 1.0 this means that there is no real thread-safety being taken care of. The newly added functionality has been added to the base 1.0 component in a way that is architecturally almost seamless and thus it adds no new issues.
Search functionality is a read functionality and thus affects no data (of course it could be affected by other threads but we do not concern ourselves with this scenario here)

2. Environment Requirements

2.1 Environment

- Development language: Java1.4
- Compile target: Java1.4, Java1.5

2.2 TopCoder Software Components

- **IDGenerator 3.0**
  - **Used for generation of unique IDs for all DataObjects that are not dependent on any database.**
- **ConfigManager 2.1.4**
  - **Used to maintain properties for mapping DataObjects to their DAO implementations and the name and namespaces to obtain Connections from the DBConnection Factory.**
- **DBConnection Factory 1.0**
  - **Used for creation of Connections for each CRUD operation.**
- **BaseException**
  - **Provides exception chaining in a Java 1.3 environment.**
- **Type Safe Enum Version 1.0**
  - **Used to enumerate the TimeEntry search types available**

NOTE: The default location for TopCoder Software component jars is../lib/tcs/COMPONENT_NAME/COMPONENT_VERSION relative to the component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location.

2.3 Third Party Components

The assumption is made that the database will be set up. Please see section 3.3 for DDL for the supported tables.

- **Jdbc driver for mysql**

mysql-connector-java-3.1.8-bin.jar

- **Jdbc driver for Informix**

ifxjdbc.jar, ifxjdbcx.jar, ifxlang.jar, ifxlsupp.jar, ifxsqlj.jar, ifxtools.jar

- **Xerces for configuration manager**

xerces.jar

3. Installation and Configuration

3.1 Package Names

com.cronos.timetracker.entry.time
com.cronos.timetracker.entry.time.search

3.2 Configuration Parameters

The configuration expects a mapping of a fully-qualified name of a DataObject type to a fully-qualified name of a target DAO and the connection name to be used with the DBConnection Factory. The namespace used in the DBConnection Factory will be the same as the one used in Config Manager.

Parameter	Description
class	The fully-qualified name of the DAO class
connectionName	Name of the connection to get from the DBConnection Factory

Here's a sample configuration file incorporating the DAOs in this design:

```
<?xml version="1.0" encoding="UTF-8"?>
<CMConfig>
  <Config name="com.topcoder.timetracker.entry.time.myconfig">
      <Property name="com.topcoder.timetracker.entry.time.TimeEntry">
                  <Property name="class">

      <Value>com.topcoder.timetracker.entry.time.TimeEntryDAO</Value>
                  </Property>
                  <Property name="connectionName">
                        <Value>informix</Value>
                  </Property>
            </Property>
            <Property name="com.topcoder.timetracker.entry.time.TaskType">
                  <Property name="class">

      <Value>com.topcoder.timetracker.entry.time.TaskTypeDAO</Value>
                  </Property>
                  <Property name="connectionName">
                        <Value>informix</Value>
                  </Property>
            </Property>
            <Property name="com.topcoder.timetracker.entry.time.TimeStatus">
                  <Property name="class">

      <Value>com.topcoder.timetracker.entry.time.TimeStatusDAO</Value>
                  </Property>
                  <Property name="connectionName">
                        <Value>informix</Value>
                  </Property>
            </Property>
        <Property name="connections">
          <!--
            The "default" property refers to a configured connection.
          -->
          <Property name="default">
```

```
                <Value>informix</Value>
            </Property>
            <!--
                The following property configures the ConnectionProducer obtaining the
Connections
                from a JDBC URL
            -->
            <Property name="informix">
                <Property name="producer">

<Value>com.topcoder.db.connectionfactory.producers.JDBCConnectionProducer</Value>
                </Property>
                <Property name="parameters">
                    <Property name="jdbc_driver">
                        <Value>com.informix.jdbc.IfxDriver</Value>
                    </Property>
                    <Property name="jdbc_url">
                        <Value>jdbc:informix-
sqli://192.168.14.219:1527/test:INFORMIXSERVER=myserver</Value>
                    </Property>
                    <Property name="user">
                        <Value>informix</Value>
                    </Property>
                    <Property name="password">
                        <Value>informix</Value>
                    </Property>
                </Property>
            </Property>
        </Config>

        <Config name="com.topcoder.db.connectionfactory.DBConnectionFactoryImpl">
            <Property name="connections">
                <!--
                    The "default" property refers to a configured connection.
                -->
                <Property name="default">
                    <Value>informix</Value>
                </Property>
                <!--
                    The following property configures the ConnectionProducer obtaining the
Connections
                    from a JDBC URL
                -->
                <Property name="informix">
                    <Property name="producer">
```

```
<Value>com.topcoder.db.connectionfactory.producers.JDBCConnectionProducer</Value>
                </Property>
                <Property name="parameters">
                   <Property name="jdbc_driver">
                      <Value>org.gjt.mm.mysql.Driver</Value>
                   </Property>
                   <Property name="jdbc_url">
                      <Value>jdbc:mysql://localhost:3306/test</Value>
                   </Property>
                   <Property name="user">
                      <Value>root</Value>
                   </Property>
                   <Property name="password">
                      <Value></Value>
                   </Property>
                </Property>
             </Property>
          </Property>
   </Config>
</CMConfig>
```

3.3 Dependencies Configuration

3.3.1 IDGenerator

In order to set up the actual sequences required by the time entry component, the developer will use the SQL insert statement defined in the IDGenerator component specification section 4.3 and use the following names as keys:

com.cronos.timetracker.entry.time.TimeEntry
com.cronos.timetracker.entry.time.TaskType
com.cronos.timetracker.entry.time.TimeStatus

The other parameters can remain as is.

The sample sql is under test_files/ IDGenerator.sql

3.3.2 DDL for tables

```
CREATE TABLE TimeEntries (
    TimeEntriesID      integer NOT NULL,
    TaskTypeID         integer, NOT NULL,
    TimeStatusesID     integer, NOT NULL,
    Description        varchar(64) NOT NULL,
    EntryDate          datetime year to second NOT NULL,
```

```
    Hours            float NOT NULL,
    Billable         smallint NOT NULL,
    CreationDate     date NOT NULL,
    CreationUser     varchar(64) NOT NULL,
    ModificationDate  date NOT NULL,
    ModificationUser  varchar(64) NOT NULL,
    PRIMARY KEY (TimeEntriesID)
);
CREATE TABLE TaskTypes (
    TaskTypesID      integer NOT NULL,
    Description      varchar(64) NOT NULL,
    CreationDate     date NOT NULL,
    CreationUser     varchar(64) NOT NULL,
    ModificationDate  date NOT NULL,
    ModificationUser  varchar(64) NOT NULL,
    PRIMARY KEY (TaskTypesID)
);
CREATE TABLE TimeStatuses (
    TimeStatusesID   integer NOT NULL,
    Description      varchar(64) NOT NULL,
    CreationDate     date NOT NULL,
    CreationUser     varchar(64) NOT NULL,
    ModificationDate  date NOT NULL,
    ModificationUser  varchar(64) NOT NULL,
    PRIMARY KEY (TimeStatusesID)
);
CREATE TABLE time_reject_reason (
    TimeEntriesID      integer NOT NULL,
    reject_reason_id   integer NOT NULL,
    creation_date      datetime year to second NOT NULL,
    creation_user      varchar(64) NOT NULL,
    modification_date  datetime year to second NOT NULL,
    modification_user  varchar(64) NOT NULL,
    PRIMARY KEY (TimeEntriesID, reject_reason_id)
);
CREATE TABLE reject_reason (
    reject_reason_id   integer NOT NULL,
    description        varchar(64) NOT NULL,
    creation_date      datetime year to second NOT NULL,
    creation_user      varchar(64) NOT NULL,
    modification_date  datetime year to second NOT NULL,
    modification_user  varchar(64) NOT NULL,
    PRIMARY KEY (reject_reason_id)
);
```

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

1) Add the jars to the classpath
2) Insert predefined items.
    a. Task Types
        - Component Specification
        - Component Design
        - Component Development
        - Information Architecture
        - Project Management
        - Meeting
        - Sales
        - Miscellaneous
    b. Time Status
        - Pending Approval – the time entry is pending approval by an administrator or supervisor
        - Approved – the time entry has been approved by an administrator or supervisor
        - Not Approved – the time entry is not approved by an administrator or supervisor

4.3 Demo

Each DAO implementation and the related entity DataObject will be shown using all CRUD methods and some DataObject getters and setters. In addition, a complete local transaction scenario will be shown involving all DAOs.

All scenarios use the configuration example shown in section 3.2. Also assume that the DBConection Factory is configured to return a Connection from the used connection name "informix" and namespace "*com.topcoder.timetracker.entry.time.myconfig*".
Finally, assume that the database has been set up and all tables created.

4.3.1 CRUD with TimeEntry and TimeEntryDAO

```
// Create a TimeEntry
TimeEntry entry = new TimeEntry();
entry.setDescription("Coding class zec");
entry.setDate(new Date());
entry.setHours(2.5F);
entry.setBillable(true);
entry.setTaskTypeId(0);
```

```
        entry.setTimeStatusId(0);

        // Get the DAO
        DAO myDAO = DAOFactory.getDAO(TimeEntry.class,
"com.topcoder.timetracker.entry.time.myconfig");

        // Create a record. assume key generated = 1
        myDAO.create(entry, "ivern");

        // update some info
        entry.setHours(3.5F);
        myDAO.update(entry, "ivern");
        System.out.println(entry.getDescription());

        // to do searches:
        // formulate a where clause
        String whereClause = "CreationUser = \'ivern\'";

        // Get TimeEntries for a range of values
        List entries = myDAO.getList(whereClause);
        System.out.println(((TimeEntry) entries.get(0)).getCreationUser());
        System.out.println(((TimeEntry) entries.get(0)).getModificationUser());

        // Or, get a specific TimeEntry and see some values
        int id = entry.getPrimaryId();
        TimeEntry myEntry = (TimeEntry) myDAO.get(id);
        System.out.println(myEntry.getDescription());
        System.out.println(myEntry.getHours());
        System.out.println(myEntry.isBillable());
        System.out.println(myEntry.getModificationUser());

        // to do deletes:
        // Delete the record with the given Id
        myDAO.delete(id);
        assertEquals(0, myDAO.getList(null).size());
```

4.3.2 CRUD with TaskType and TaskTypeDAO

```
            // Create a TaskType
            TaskType type = new TaskType();
            type.setDescription("Component Specification");

            // Get the DAO
            DAO myDAO = DAOFactory.getDAO(TaskType.class,
        "com.topcoder.timetracker.entry.time.myconfig");
```

```
// Create a record.
myDAO.create(type, "ivern");

// update some info
type.setDescription("Component Spellification");
myDAO.update(type, "TangentZ");
System.out.println(type.getDescription());

// to do searches:
// formulate a where clause
String whereClause = "CreationUser = \'ivern\'";

// Get TaskTypes for a range of values
List types = myDAO.getList(whereClause);
assertEquals(1, types.size());
System.out.println(((TaskType) types.get(0)).getCreationUser());
System.out.println(((TaskType) types.get(0)).getModificationUser());

// Or, get a specific Task Type and see some values
int id = type.getPrimaryId();
TaskType myType = (TaskType) myDAO.get(id);

// to do deletes:
// Delete the record with the given Id
myDAO.delete(id);
assertEquals(0, myDAO.getList(null).size());
```

4.3.3 CRUD with TimeStatus and TimeStatusDAO

```
// Create a TimeStatus
TimeStatus status = new TimeStatus();
status.setDescription("Reaby");

// Get the DAO
DAO myDAO = DAOFactory.getDAO(TimeStatus.class,
"com.topcoder.timetracker.entry.time.myconfig");

// Create a record.
myDAO.create(status, "ivern");

// update some info
status.setDescription("Ready");
myDAO.update(status, "TangentZ");
System.out.println(status.getDescription());

// to do searches:
```

```
// formulate a where clause
String whereClause = "CreationUser = \'ivern\'";

// Get TimeStatuses for a range of values
List statuses = myDAO.getList(whereClause);
assertEquals(1, statuses.size());
System.out.println(((TimeStatus) statuses.get(0)).getCreationUser());
System.out.println(((TimeStatus) statuses.get(0)).getModificationUser());

// Or, get a specific TimeStatus and see some values
int id = status.getPrimaryId();
TimeStatus myStatus = (TimeStatus) myDAO.get(id);

// to do deletes:
// Delete the record with the given Id
myDAO.delete(id);
assertEquals(0, myDAO.getList(null).size());
```

4.3.4 Using with local transaction control.

```
// obtain DAOs for entities
DAO myTimeStatusDAO = DAOFactory.getDAO(TimeStatus.class,
NAMESPACE);
DAO myTaskTypeDAO = DAOFactory.getDAO(TaskType.class,
NAMESPACE);

// set connections in all DAOs for local transaction. assume connection
exists and is ready
// for local transaction (autoCommit=false);
myTimeStatusDAO.setConnection(conn);
myTaskTypeDAO.setConnection(conn);

// initiate local transaction
// create new taskType
TaskType type = new TaskType();
type.setDescription("Component Devastation");
myTaskTypeDAO.create(type, CREATION_USER);
System.out.println(type.getModificationUser());

// delete old taskType
int taskTypeId = type.getPrimaryId();
myTaskTypeDAO.delete(taskTypeId);
assertEquals(0, myTaskTypeDAO.getList(null).size());

// create new timeStatus
TimeStatus status = new TimeStatus();
```

```
status.setDescription("ready");
myTimeStatusDAO.create(status, CREATION_USER);
System.out.println(status.getModificationUser());

// update the timeStatus to new status
status.setDescription("down");
myTimeStatusDAO.update(status, MODIFICATION_USER);
System.out.println(status.getDescription());

// remove connections
myTimeStatusDAO.removeConnection();
myTaskTypeDAO.removeConnection();
```

4.3.5 Managing Synchronous Batch operations

```
// Get a time entry DAO
DAO myDAO = DAOFactory.getDAO(TimeEntry.class
                              ,"com.topcoder.timetracker.entry.time.myconfig");

// create a number of TimeEntry objects and
// put then in an array
DataObject[] dataToCreate = new DataObject[…];
. . .

//////////////////////////////////////////////////////////////////////
// do a batch create. First we create a ResultData Object where we want
// the results to be deposited.
ResultData myResults = new ResultData();
// do the batch, do NOT make it all or nothing
try{
        myDAO.batchCreate(dataToCreate
                          , “Ivern”
                          , false
                          , myResults);
}catch(BatchOperationException boe){
        // we had some serious failure
        // inspect
}

//
// we can now inspect the result to see how things went

// how many failures
System.out.println(“ we had “
        + myResults.getFailedCount()
        + “ failures”);
// how many succeeded
```

```
System.out.println(“ we had “
        + myResults.getSuccessCount()
        + “ successful operations”);

// Printout all the failed operations
BatchOperationException[] exceptions =
        myResults.getExceptionList();
Object[] operations = myResults.getOperations();
// determine operation result
for (int i = 0; i < exceptions.length; i++){
        if(exceptions[i] == null){
        // statement was successful. No exception
        ...
        }else{
        // error on statement
        System.put.println(“ For the following create operation: “
                + ((TimeEntry)(operations[i])).toString());
                System.put.println(“ We had the following failure: “
                + exceptions[i].toString());
    }
}
////////////////////////////////////////////////////////////////////////
// do a batch read
myResults = new ResultData();
// create a number of TimeEntry ids to read
// put then in an array
long[] dataToget = new long[…];
        . . .
// do the batch, this time we will make it all or nothing
try{
        myDAO.batchRead(dataToget, true, myResults);
}catch(BatchOperationException boe){
        // we had some serious failure
        // inspect
}
// get the results
TimeEntry[] results = (TimeEntry[])myResults.getBatchResults();
```

4.3.6 Managing Synchronous Batch operations

```
// we will create a wrapper around our existing batch enabled dao
AsynchBatchDAOWrapper myAsynchDAO=
        new AsynchBatchDAOWrapper (myDAO);
// we will set the connection that we are using to autocommit for
// more reliable processing, but it is suggested that when dealing with
// batch processing the user actually do not use their own connection.
myDAO.getConnection().setAutoCommit( false );
```

```
// create some dummy data for updates
DataObject[] updateData = new DataObject[…];
. . .
// Before we can do a batch update we will create a listener for our
// notification results. We will use an anonymous class
ResultListener myListener = new ResultListener(){
        public void notify(ResultData results){
                // Process the results here
        }};

// do a batch update
int threadId = myAsynchDAO.asynchBatchUpdate(updateData
                        , “Ivern”
                        , true
                        , myListener);
// note that we continue here with other stuff.
. . .
```

4.3.7 Reject reason operations

```
// We create a new rejection reason and set its values
// before we add it to a time entry
RejectReason reason1 = new RejectReason();
reason1.setDescription(“20 hours for a demo? Don’t think so”);
// add the reason to some time entry
timeEntry.addRejectReason(reason1);
// persist the entry
myDAO.create(myDAO, “Ivern”);

// get all the rejection reasons for some time entry
TimeEntry myEntry = (TimeEntry)(myDAO.get(15426));
RejectReason[] reasons = myEntry.getAllRejectReasons();
// or we can ask for only one reason if we have the id
RejectReason someReason = myEntry.getRejectReason(768009);
// remove a specific Reject reason
myEntry.removeRejectReason(768009);
// update the time entry, the reject reason should be removed
myDAO.update(myEntry, “Ivern”);
myEntry = (TimeEntry)(myDAO.get(15426));
// this should not contain the removed entry
reasons = myEntry.getAllRejectReasons();
```

4.3.8 Search examples

```
// create an instance of TimeEntryDAO
// Get the DAO
DAO myDAO = DAOFactory.getDAO(TimeEntry.class
        , "com.topcoder.timetracker.entry.time.myconfig");
```

```
//
//get all time entries that contain the description “Doomed Project”
SearchExpression expression1 = SubstringExpression.contains(
                                        TimeEntryCriteria.DESCRIPTION,
                                        “Doomed Project”);
// do the actual search
ExpressionEvaluator evaluator =
        new SQLBasedTimeEntryCriteriaExpressionEvaluator(myDAO);
TimeEntry[] matches = (TimeEntry[])(evaluator.evaluate(expression1));

//
// get all time entries that contain the statusID of “10000”
SearchExpression expression2 = ComparisonExpression.equals(
                        TimeEntryCriteria.TIME_STATUS_ID
                        , “10000”);
matches = (TimeEntry[])(evaluator.evaluate(expression2));

// We can do a reverse and by using a boolean NOT we can actual get all
// that do NOT contain this status id
SearchExpression expression2NOT =
BooleanExpression.not(expression2);
matches = (TimeEntry[])(evaluator.evaluate(expression2NOT));

// Search by task types
SearchExpression expression3 = ComparisonExpression.equals(
                        TimeEntryCriteria.TASK_TYPE_ID
                        , “20000”);
matches = (TimeEntry[])(evaluator.evaluate(expression3));

//
// search based on users, Here we will create an OR boolean query
// Get all the time entries that were created OR modified by a user
SearchExpression expression4_left =
        ComparisonExpression.equals(
                TimeEntryCriteria.CREATION_USER
                , “Ivern”);
SearchExpression expression4_right =
        ComparisonExpression.equals(
                TimeEntryCriteria.MODIFICATION_USER
                , “Ivern”);
SearchExpression expression4OR =
        BooleanExpression.or(expression4_left, expression4_right);
matches = (TimeEntry[])(evaluator.evaluate(expression4OR));

//
```

```
// Get all billable entries
SearchExpression expression5 =
        ComparisonExpression.equals(
                TimeEntryCriteria.BILLABLE_FLAG,
                , “false”);
matches = (TimeEntry[])(evaluator.evaluate(expression5));

//
// Get entries based on reject reason id
SearchExpression expression6 =
        ComparisonExpression.equals(
                TimeEntryCriteria.REJECT_REASON_ID,
                , “30000”);
matches = (TimeEntry[])(evaluator.evaluate(expression6));

//
// Get entries based on hour ranges
SearchExpression expression7 =
        RangeExpression.fromTo(TimeEntryCriteria.HOURS
                                , “5”
                                , “10”);
matches = (TimeEntry[])(evaluator.evaluate(expression7));

//
// we can also do this based on ComparisonExpression
SearchExpression expression8a =
        ComparisonExpression.greaterThan(TimeEntryCriteria.HOURS
                                                , “10”);
matches = (TimeEntry[])(evaluator.evaluate(expression8a));
        SearchExpression expression8b =
ComparisonExpression.lessThanOrEqual(
                        TimeEntryCriteria.HOURS
                        , “10”);
matches = (TimeEntry[])(evaluator.evaluate(expression8a));

//
// get record based on creation date (modification date is the same)
SearchExpression expression9a =
        ComparisonExpression.greaterThan(
                        TimeEntryCriteria.CREATION_DATE
                        , “01/01/2006”);
matches = (TimeEntry[])(evaluator.evaluate(expression9a));
SearchExpression expression9b =
        ComparisonExpression.lessThanOrEqual(
                        TimeEntryCriteria.CREATION_DATE
                        , “01/01/2006”);
```

```
matches = (TimeEntry[])(evaluator.evaluate(expression9b));
SearchExpression expression9c =
        BooleanExpression.and(
                ComparisonExpression.greaterThan(
                        TimeEntryCriteria.CREATION_DATE
                        , “01/01/2006”)
                , ComparisonExpression.lessThan(
                        TimeEntryCriteria.CREATION_DATE
                        , “02/28/2006”));
matches = (TimeEntry[])(evaluator.evaluate(expression9a));
```

5. Future Enhancements

This component is very open to future enhancements.

1) Add more robust persistence methods to the DAO, such as the ability to do batch creates, deletes, etc.
2) Create a more user-friendly ability to add where clauses to sql statements or XPath expressions for XML searches.
3) Add concept of a dirty field to each DataObject in order to maintain which fields are actually modified before updating the record. The current iteration calls for all fields to be updated, whether they were actually modified or not. Future iterations could narrow the list of updated fields to just ones that were actually modified.
4) Paging can be added to the getList method, so a paged subset would be returned.
5) Create a façade for easier manipulation of entities. Current design calls for manipulation of entities using their primary ids. The façade could hide this. For example, the task type in a TimeEntry could be updated by passing the TaskType object and letting the façade figure out which Id it needs to use.