Time Entry 1.0 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.

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.

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.

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, TaskTypeID, TimeStatusID,
Description, Date, 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:
setLong(1, TimeEntry.primaryId)
setLong(2, TimeEntry.taskTypeId)
setLong(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:
setLong(1, TaskType.primaryId)
setString(2, TaskType.description)
setString(3, user)
setDate (4, date)
setString (5, user)
setDate (6, date)

TimeStatusDAO:
setLong(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 TaskTypeID=?, TimeStatusID=?, Description=?, Date=?, 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:
setLong(1, TimeEntry.taskTypeId)
setLong(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)
setLong(9, TimeEntry.primaryId)

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

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

getLong("TimeEntriesID")	→ TimeEntry.primaryId
getLong("TaskTypeID")	→ TimeEntry.taskTypeId
getLong("TimeStatusesID")	→ TimeEntry.timeStatusId
getString("Description")	→ TimeEntry.description
getDate("Date")	→ 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:

getLong("TaskTypesID")	→ TaskType.primaryId
getString("Description")	→ TaskType.description
getString("CreationUser")	→ TaskType.creationUser
getDate ("CreationDate")	→ TaskType.creationDate
getString ("ModificationUser")	→ TaskType.modificationUser
getDate ("ModificationDate")	→ TaskType.modificationDate

TimeStatusDAO:

getLong("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:

getLong(“TimeEntriesID”)	→ TimeEntry.primaryId
getLong(“TaskTypeID”)	→ TimeEntry.taskTypeId
getLong(“TimeStatusesID”)	→ TimeEntry.timeStatusId
getString(“Description”)	→ TimeEntry.description
getDate(“Date”)	→ 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:

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

TimeStatusDAO:

getLong(“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 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.

1.3.9 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 behaviour. 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.

1.4 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 operations and three methods that can be used to external manage connection for the DAO if local trasactions are desired. For non-JDBC DAOs, it is recommended that these three methods throw UnsupportedOperationException.

BaseDAO:
Base data access component. Implements all methods in the DAO interface. The five CRUD operations are implemented, but each method relies on helper methods to provide entity-specific information. Most of these methods are not implemented and declared abstract.

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.

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:

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.

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

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.

2. Environment Requirements

2.1 Environment

- At minimum, Java 1.3 is required for compilation and executing test cases.

2.2 TopCoder Software Components

- **IDGenerator 2.0**
    - **Used for generation of unique IDs for all DataObjects that are not dependent on any database.**
- **ConfigManager 2.1.3**
    - **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.**

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.

3. Installation and Configuration

3.1 Package Names

com.topcoder.timetracker.entry.time

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:

```
<CMConfig>
    <Config name="com.topcoder.entry.time.myconfig">
        <Property name="com.topcoder.entry.time.TimeEntry">
            <Property name="class">
                <value>com.topcoder.entry.time.TimeEntryDAO</value>
            </Property>
            <Property name="connectionName">
                <value>"informix"</value>
            </Property>
        </Property>
        <Property name="com.topcoder.entry.time.TaskType">
            <Property name="class">
                <value>com.topcoder.entry.time.TaskTypeDAO</value>
            </Property>
            <Property name="connectionName">
                <value>"informix"</value>
            </Property>
        </Property>
        <Property name="com.topcoder.entry.time.TimeStatus">
            <Property name="class">
                <value>com.topcoder.entry.time.TimeStatusDAO</value>
            </Property>
            <Property name="connectionName">
                <value>"informix"</value>
            </Property>
        </Property>
    </Config>
</CMConfig>
```

3.3 Dependencies Configuration

3.3.1 IDGenerator

The developer should follow the IDGenerator component specification to configure the component.

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

The other parameters can remain as is.

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,
    Date                date 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)
);
```

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.tocoder.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(1L);
entry.setTimeStatusId(1L);
```

```
// Get the DAO
DAO myDAO = DAOFactory.getDAO(TimeEntry.class,"
com.tocoder.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");


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

// Get TimeEntries for a range of values
List entries = myDAO.getList(whereClause);

// Or, get a specific TimeEntry and see some values
TimeEntry myEntry = (TimeEntry)myDAO.get(1L);
System.out.println(myEntry.getDescription());
System.out.println(myEntry.getHours());
System.out.println(myEntry.getBillable());
System.out.println(myEntry.getModificationUser());

// to do deletes:
// Delete the record with the given Id
myDAO.delete(1L);
```

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.tocoder.entry.time.myconfig");

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

// update some info
type.setDescription("Component Spellification");
myDAO.update(type,"ivern");
```

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

// Get TaskTypes for a range of values
List types = myDAO.getList(whereClause);

// Or, get a specific Task Type and see some values
TaskType myType = (TaskType)myDAO.get(1L);
System.out.println(myType.getDescription());
System.out.println(myType.getModificationUser());


// to do deletes:
// Delete the record with the given Id
myDAO.delete(1L);
```

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.tocoder.entry.time.myconfig");

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

// update some info
status.setDescription("Ready");
myDAO.update(status,"ivern");


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

// Get TimeStatuses for a range of values
List statuses = myDAO.getList(whereClause);

// Or, get a specific TimeStatus and see some values
TimeStatus myStatus = (TimeStatus)myDAO.get(1L);
System.out.println(myStatus.getDescription());
System.out.println(myStatus.getModificationUser());
```

```
// to do deletes:
// Delete the record with the given Id
myDAO.delete(1L);
```

4.3.4 *Using with local transaction control.*

```
// obtain DAOs for entities
DAO myTimeEntryDAO = DAOFactory.getDAO(TimeEntry.class,"
com.tocoder.entry.time.myconfig");
DAO myTaskTypeDAO = DAOFactory.getDAO(TaskType.class,"
com.tocoder.entry.time.myconfig");

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

// initiate local transaction

// create new status
TaskType type = new TaskType();
type.setDescription("Component Devastation");
myTaskTypeDAO.create(type);
long taskTypeId = type.getPrimaryId();

// delete old status whose id is 1
myTaskTypeDAO.delete(1L);

// update a time entry to new status
TimeEntry entry = (TimeEntry)myTimeEntryDAO.get(1L);
entry.setTaskTypeId(taskTypeId);
myTimeEntryDAO.update(entry);

// commit
connection.commit();

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

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 tsk type in a TimeEntry could be updated by passing the TaskType object and letting the façade figure out which Id it needs to use.