Expense Entry 1.1 Component Specification

1. Design

The Expense Entry custom component is part of the Time Tracker application. It provides an abstraction of an expense entry that an employer enter into the system on a regular basis. This component handles the persistence and other business logic required by the application.

The manager classes handle the business logic. They provide a façade for the user to the functionality offered by this component: expense entry management, expense entry types management, expense entry statuses management.

The following classes implement the model part of this application: ExpenseEntry, ExpenseEntryType and ExpenseEntryStatus.

The persistence layer is abstracted using an interface. This allows easy swapping of the persistence storage without changes to the rest of the component. The default implementation uses an Informix database for storage.

Version 1.1 adds a few additional functionalities:

- bulk operations on expense entries
- attaching zero or more reject reasons to an expense entry
- a powerful expense entry search framework

The reject reason functionality is implemented by adding a list of reject reasons and the appropriate modifying and retrieval methods to the ExpenseEntry data class. The list contains instances of the ExpenseEntryRejectReason data class. The data class is needed because the reject reasons linked to an expense entry are characterized by a creation and modification user and date. In order to accommodate the ExpenseEntryRejectReason class into the data class hierarchy, CommonInfo had to be split in two. It was done because ExpenseEntryRejectReason doesn't have an id or a description. These attributes and the corresponding setters and getters have been moved to BasicInfo (extends CommonInfo). The old data classes extend from BasicInfo thus having the same API as before. The new data class extends from CommonInfo thus having only what it needs.

The bulk operations on expense entries are implemented at the ExpenseEntryManager, ExpenseEntryPersistence and ExpenseEntryDbPersistence levels. All these classes and interfaces have four new methods (for bulk adding, deleting, updating and retrieving). However, only ExpenseEntryDbPersistence has significant business logic. The bulk operations accept an array of ExpenseEntry objects or expense entry ids. They also have an isAtomic flag. An atomic bulk operation means that the entire operation fails if a single operation fails (in which case no changes are done to the DB and no results are returned). Non-atomic means each operation is treated individually. If one operation fails, the others are processed anyway. An array is returned to the user with those expense entries or ids that have caused problems.

The expense entry search framework is very powerful. It is built around a strategy pattern. This means that additional search criteria can be added easily. The Criteria interface abstracts the search criteria. It is oriented towards DB implementations and it is used to perform the filtering at the database level (without bringing all expense entries into memory) for obvious performance reasons. The Criteria has methods for returning the where clause expression to be used for filtering. Since it can have parameters, another method returns them (they are not embedded in the text expression for portability reasons; a PreparedStatement is used for insertion). The implementations of the Criteria interface contain a perfect field match filter, a like (SQL style) field match filter, a between (SQL style) field match filter and a special RejectReasonCriteria (because it uses the

exp_reject_reason table it has a special where clause that uses “IN (SELECT …)” or “EXISTS”). There are also two aggregate filters, NotCriteria and CompositeCriteria that provide the usual boolean operations (NOT, AND, OR). Note that many implementations are very generic (they can express a constraint on any field). Constants and shortcut methods for the explicitly required filters are provided.

Useful additional features:

- any perfect match queries can be performed (not just on the required fields)
- any SQL like match queries can be performed (not just on the required fields)
- any SQL between and <=, => match queries can be performed (not just on the required fields)

1.1 Design Patterns

The **strategy** pattern is used for abstracting the persistence implementation in the ExpenseEntryPersistence interface. It is also used in the Criteria interface and its subclasses to abstract the search filtering criteria.

The **data value object** pattern is used in the ExpenseEntry, ExpenseEntryType, ExpenseEntryStatus and ExpenseEntryRejectReason classes. These classes facilitate the data exchange between the business logic and the persistence layer.

The **composite pattern** is used in the CompositeCriteria class. A CompositeCriteria object aggregates other Criteria instances, including even instances of CompositeCriteria itself.

The search classes have **factory methods** for creating the filters from the requirements.

1.2 Industry Standards

SQL, JDBC

1.3 Required Algorithms

SQL statements are very simple select, insert, update and delete queries so, it is not necessary to provide them as algorithms.
I will provide an algorithm for the constructor of ExpenseEntryManager(the algorithms for the constructors oh the other two managers are almost identical).
The algorithm is:

```
//read the configuration file
ConfigManager cm = ConfigManager.getInstance();
String className = cm.getStringArray(namespace,"entry_persistence_class");
String connectionProducerName = cm.getStringArray(namespace,"
                                                    connection_producer_name");

//create the persistence instance through reflection

Class c = Class.forName(className);
if(connectionProducerName == null) {
      persistence = (ExpenseEntryPersistence)c.newInstance();
} else {
       Constructor constr = c.getConstructor(new Class[]{Class.forName(“String”});
       persistence = (ExpenseEntryPersistence) constr.newInstance(new
                                                  Object[]{connectionProducerName)});
}
```

1.3.1 Composite and reject reason criteria

These two criteria might present some difficulties in implementation. The composite criteria works by concatenating the where clauses of the contained criteria:

- result = empty string
- for each contained criteria
- if not first criteria, result = result + compositionKeyword
- for all criteria, result = result + "(" + criteria.getWhereClause() + ")"
- return result

The parameters are obtained by concatenating the parameters of the contained criteria:

- result = empty ArrayList
- iterates the contained criteria
- for each criteria, calls getParameters()
- accumulate each result to the result list (in the exact order in which they are obtained from the contained criteria)
- return the result

The reject reason criteria is simple but it uses IN SELECT which some people might not know. So the where clause is obtained using:

- "? IN (SELECT reject_reason_id FROM exp_reject_reason "
  + "WHERE exp_reject_reason.ExpenseEntriesId = expenseEntries.ExpenseEntriesId)".

Other ways can be used as well:

- EXISTS (SELECT reject_reason_id FROM exp_reject_reason "
  + "WHERE exp_reject_reason.ExpenseEntriesId = expenseEntries.ExpenseEntriesId"
  + "AND exp_reject_reason.reject_reason_id = ?)".

In fact the second might be better. Some databases might not like the fact that IN is used with a value (not a field) in the first case.

1.4 Component Class Overview

ExpenseEntry

This class holds the information about an expense entry. In addition to common information, an expense entry also contains the date, amount of money, the type, the current status and a flag indicating whether the client should be billed.

Version 1.1 also adds a rejectReason list containing ExpenseEntryRejectReason objects. This list contains the reject reasons for the expense entry. The list has a range of accessory methods for consulting and changing it.

When creating an instance of this class the user has two options:

1) Use the default constructor and allow the GUID Generator component to generate a unique id

2) Use the parameterized constructor and provide an id for the ExpenseEntry instance; if the id already is contained by another entry from the ExpenseEntries table, then the newly created entry will not be added to the ExpenseEntries table. Also the user should not populate the creationDate and modificationDate fields, because if he does, the entry will not be added to the database. This fields will be handled automatically by the component (the current date will be used). When loading from the persistence, all the fields will be properly populated.

Thread safety:

Because the class is mutable it is not thread safe. Threads will typically not share instances but if they do, the mutability should be used with care since any change done in one thread will affect the other thread, possibly without even being aware of the change.

ExpenseEntryType

This class holds the information about an expense entry type.When creating an instance of this class the user has two options:

1) Use the default constructor and allow the GUID Generator component to generate a unique id

2) Use the parameterized constructor and provide an id for the ExpenseEntryType instance; if the id already is contained by another type from the ExpenseTypes table, then the newly created type will not be added to the ExpenseTypes table.

Also the user should not populate the creationDate and modificationDate fields, because if he does, the type will not be added to the database. These fields will be handled automatically by the component (the current date will be used). When loading from the persistence, all the fields will be properly populated.

ExpenseEntryStatus

This class holds the information about an expense entry status.When creating an instance of this class the user has two options:

1) Use the default constructor and allow the GUID Generator component to generate a unique id

2) Use the parameterized constructor and provide an id for the ExpenseEntryStatus instance; if the id already is contained by another status from the ExpenseStatuses table, then the newly created status will not be added to the ExpenseStatuses table.

Also the user should not populate the creationDate and modificationDate fields, because if he does, the status will not be added to the database. This fields will be handled automatically by the component (the current date will be used). When loading from the persistence, all the fields will be properly populated.

ExpenseEntryRejectReason

This is a data class for representing an expense entry to reject reason id association (maps to the exp_reject_reason) table. This class is needed because the association has the creation and modification dates and users fields. Without those, the ExpenseEntry class could have easily been storing the reject reason ids directly.

This class stores the rejectReasonId and inherits the date and user fields from the CommonInfo class.

Thread safety:

Because the class is mutable it is not thread safe. Threads will typically not share instances but if they do, the mutability should be used with care since any change done in one thread will affect the other thread, possibly without even being aware of the change.

CommonInfo

This abstract contains the common features of ExpenseEntry, ExpenseEntryType, ExpenseEntryStatus and ExpenseEntryRejectReason classes. This class is abstract because there is no need to instantiate it directly. It has no abstract methods; it is made abstract to group the common features of the data classes and to prevent direct instantiation. This abstract class implements the Serializable interface, so all the concrete implementations of this class will be serializable.

The class contains the creation and modification dates and users. Note that id and description have been moved to BasicInfo. That's because ExpenseEntryRejectReason

doesn't have them. The ExpenseEntry, ExpenseEntryType and ExpenseEntryStatus classes extend now from BasicInfo.

Thread safety:

Because the class is mutable it is not thread safe. Threads will typically not share instances but if they do, the mutability should be used with care since any change done in one thread will affect the other thread, possibly without even being aware of the change.

BasicInfo
This abstract contains the common features of ExpenseEntry, ExpenseEntryType and ExpenseEntryStatus classes.

This class is abstract because there is no need to instantiate it directly. It has no abstract methods; it is made abstract to group the common features of the data classes and to prevent direct instantiation.

Note that id and description have been moved from CommonInfo. That's because ExpenseEntryRejectReason doesn't have them. The rest of the attributes/accessors needed by the three subclasses are inherited from CommonInfo.

Thread safety:

Because the class is mutable it is not thread safe. Threads will typically not share instances but if they do, the mutability should be used with care since any change done in one thread will affect the other thread, possibly without even being aware of the change.

ExpenseEntryTypeManager
The ExpenseEntryTypeManager class is a facade for the types management functionality.
It can do the following things:
-add a type(ExpenseEntryType instance) to the ExpenseTypes table; if the ExpenseEntryType instance has the id=-1 this manager will use the GUID Generator to generate an id for the ExpenseEntryType instance
-delete a type from the ExpenseTypes table
-delete all the types from the ExpenseTypes table
-retrieve a type(given its id) from the ExpenseTypes table
-retrieve all the types from the ExpenseTypes table
-update a type in the ExpenseTypes table
This manager is responsible for reading two properties from the configuration file. To accomplish this it will use the
Configuration Manager component. From the configuration file two properties will be read:
-a class name identifying an implementation of ExpenseEntryTypePersistence(this property is required)
-a connection producer name identifying a ConnectionProducer instance(this property is optional). This
instance of ConnectionProducer will provide the connection to the database.
Using these two properties this manager will create thourough reflection an ExpenseEntryTypePersistence implementation instance.

ExpenseEntryStatusManager
The ExpenseEntryStatusManager class is a facade for the statuses management functionality.
It can do the following things:

-add a status(ExpenseEntryStatus instance) to the ExpenseStatuses table; if the ExpenseEntryStatus instance has the id=-1 this manager will use the GUID Generator to generate an id for the ExpenseEntryStatus instance
-delete a status from the ExpenseStatuses table
-delete all the statuses from the ExpenseStatuses table
-retrieve a status(given its id) from the ExpenseStatuses table
-retrieve all the statuses from the ExpenseStatuses table
-update a status in the ExpenseStatuses table
This manager is responsible for reading two properties from the configuration file. To accomplish this it will use the
Configuration Manager component. From the configuration file two properties will be read:
-a class name identifying an implementation of ExpenseEntryStatusPersistence(this property is required)
-a connection producer name identifying a ConnectionProducer instance(this property is optional). This instance of ConnectionProducer will provide the connection to the database. Using these two properties this manager will create thourough reflection an ExpenseEntryStatusPersistence implementation instance.

ExpenseEntryManager
The ExpenseEntryManager class is a facade for the entries management functionality.
. It can do the following things:
-add an entry(ExpenseEntry instance) to the ExpenseEntries table; if the ExpenseEntry instance has the id=-1 this manager will use the GUID Generator to generate an id for the ExpenseEntry instance
-delete an entry from the ExpenseEntries table
-delete all the entries from the ExpenseEntries table
-retrieve an entry(given its id) from the ExpenseEntries table
-retrieve all the entries from the ExpenseEntries table
-update an entry in the ExpenseEntries table
This manager is responsible for reading two properties from the configuration file. To accomplish this it will use the
Configuration Manager component. From the configuration file two properties will be read:
-a class name identifying an implementation of ExpenseEntryPersistence(this property is required)
-a connection producer name identifying a ConnectionProducer instance(this property is optional). This
instance of ConnectionProducer will provide the connection to the database. Using these two properties this manager will create thourough reflection an ExpenseEntryPersistence implementation instance.

Changes since 1.0

Four new methods for doing bulk operations on sets of expense entries have been added. These methods can work in atomic mode (a failure on one entry causes the entire operation to fail) or non-atomic (a failure in one entry doesn't affect the other and the user has a way to know which ones failed).

There is also a search method that provides the capability to search expense entries at the database level and return the ones that match.

ExpenseEntryPersistence
ExpenseEntryPersistence represents the interface for expense entries access. Client can choose between alternative implementations to suit persistence migration. Interface defines all necessary methods to interact with the database. The methods exposes by this interface are very raw (it would be hard for a user to use them to obtain the

functionality). They are aimed to an efficient database implementation (using INSERT, SELECT, UPDATE and DELETE statements) but other storage technologies can be used just as well (such as XML).

Changes since 1.0

Four new methods for doing bulk operations on sets of expense entries have been added. These methods can work in atomic mode (a failure on one entry causes the entire operation to fail) or non-atomic (a failure in one entry doesn't affect the other and the user has a way to know which ones failed).

There is also a search method that provides the capability to search expense entries at the database level and return the ones that match.

ExpenseEntryTypePersistence
ExpenseEntryTypePersistence represents the interface for expense entry types access. Client can choose between alternative implementations to suit persistence migration. Interface defines all necessary methods to interact with the database. The methods exposes by this interface are very raw (it would be hard for a user to use them to obtain the functionality). They are aimed to an efficient database implementation (using INSERT, SELECT, UPDATE and DELETE statements) but other storage technologies can be used just as well (such as XML).

ExpenseEntryStatusPersistence
ExpenseEntryTypePersistence represents the interface for expense entry types access. Client can choose between alternative implementations to suit persistence migration. Interface defines all necessary methods to interact with the database. The methods exposes by this interface are very raw (it would be hard for a user to use them to obtain the functionality). They are aimed to an efficient database implementation (using INSERT, SELECT, UPDATE and DELETE statements) but other storage technologies can be used just as well (such as XML).

ExpenseEntryDbPersistence
This class is a concrete implementation of the ExpenseEntryPersistence interface that uses an database as persistence. This implementation uses the DB Connection Factory component to obtain a connection to the database. Transaction should be employed to ensure atomicity. This class provides two constructors. The first is an empty constructor, and the second can be used to specify the connection producer name which will be used to obtain a connection. The connection will not be initialized in the constructors. It will be initialized in one of the methods that will access the database; it will be initialized the first time one of this methods is called. It can be initialized using the setter or the initConnection method.

Changes since 1.0

Four new methods for doing bulk operations on sets of expense entries have been added. These methods can work in atomic mode (a failure on one entry causes the entire operation to fail) or non-atomic (a failure in one entry doesn't affect the other and the user has a way to know which ones failed).

There is also a search method that provides the capability to search expense entries at the database level and return the ones that match.

ExpenseEntryTypeDbPersistence
This class is a concrete implementation of the ExpenseEntryTypePersistence interface that uses an database as persistence.This implementation uses the DB Connection Factory component to obtain a connection to the database. Transaction should be employed to ensure atomicity. This class provides two constructors. The first is an empty

constructor, and the second can be used to specify the connection producer name which will be used to obtain a connection. The connection will not be initialized in the constructors. It will be initialized in one of the methods that will access the database; it will be initialized the first time one of this methods is called. It can be initialized using the setter or the initConnection method.

ExpenseEntryStatusDbPersistence
This class is a concrete implementation of the ExpenseEntryStatusPersistence interface that uses an database as persistence. This implementation uses the DB Connection Factory component to obtain a connection to the database. Transaction should be employed to ensure atomicity. This class provides two constructors. The first is an empty constructor, and the second can be used to specify the connection producer name which will be used to obtain a connection. The connection will not be initialized in the constructors. It will be initialized in one of the methods that will access the database; it will be initialized the first time one of this methods is called. It can be initialized using the setter or the initConnection method.

Criteria
This interface abstracts a criteria used for searching expense entries. The criteria are database oriented, approach chosen for speed by being able to filter expense entries at the SQL query level, thus avoiding the need to bring all expense entries into memory.

The criteria implementations are used to build the where clause of an SQL query on the expense entry table. To do that, the actual clause expression (string) is needed. Since the clause may have user given parameters the interface has a method to return that too. The parameters are NOT inserted into the expression directly for portability reasons (different database implementations may represent data types in different ways). PreparedStatements are used instead.

FieldMatchCriteria
This class represents a basic type of criteria for exact matching of a field with a given value. Both the field and value are given by the user (the where clause expression will look like "field=value").

The class defines 5 constants and 5 static methods for the fields the requirements specifically mention. This is done in an effort to provide the simplest possible API for the user.

Thread safety:

Immutable class, so there are no thread safety issues.

FieldLikeCriteria
This class represents a basic type of criteria for a like type of match on a field (like as in the SQL like clause). Both the field and like pattern are given by the user (the where clause expression will be "field like pattern").

The class defines a constant and a static method for the field the requirements specifically mention (description). This is done in an effort to provide the simplest possible API for the user.

Thread safety:

Immutable class, so there are no thread safety issues.

FieldBetweenCriteria

This class represents a basic type of criteria for selecting records with a field within a given interval. The field name and the value limits are given by the user. The interval can be open ended (one of the limits can be null, interpreted as no limit). When both limits are not null, the criteria matches to an SQL between clause. When one limit is missing, the criteria is a <= or => comparison.

The class defines 3 constants and 3 static methods for the fields the requirements specifically mention. This is done in an effort to provide the simplest possible API for the user.

Thread safety:

Immutable class, so there are no thread safety issues.

RejectReasonCriteria
This class represents a very specific type of expense entry match, matching those expense entries with a given reject reason id. It is a very specific criteria because it doesn't act on the expense entry table but on the exp_reject_reason table. Because of that, the where clause is very specific and it uses the "IN (SELECT ...)" or “EXISTS” SQL nested queries.

Thread safety:

Immutable class, so there are no thread safety issues.

NotCriteria
This class represents a special type of criteria that simply negates the expression of another criteria. It contains another criteria and the where clause return method delegates the call to that criteria and surrounds the result with brackets and prefixes it with NOT. The parameters are the same as the contained criteria (since no new parameters are inserted).

Thread safety:

Immutable class, so there are no thread safety issues.

CompositeCriteria
This class represents a special type of rule that is an aggregation over two or more rules. The aggregation type can be AND and OR with the usual boolean logic significance.

The class has an attribute and a getter for the contained criteria. The keyword for the aggregation type is parameterized. However, considering the current SQL standard, only AND or OR are expected to be used (constants are static creation methods are defined for them). But maybe some database implementation might implement other aggregation types (such as XOR for example).

Thread safety:

Immutable class, so there are no thread safety issues.

1.5 Component Exception Definitions

Persistence Exception[custom]
The PersistenceException exception is used to wrap any persistence implementation specific exception. These exceptions are thrown by the persistence interfaces implementations. Since they are implementation specific, there needs to be a common way to report them to the user, and that is what this exception is used for. This exception is originally thrown in the persistence implementations. The business logic layer (the manager classes) will forward them to the user.

ConfigurationException[custom]
This exception is thrown by the managers if anything goes wrong in the process of loading the configuration file or if the information is missing or corrupt.

InsufficientDataException[custom]
This exception is thrown when some required fields (NOT NULL) are not set when creating or updating an entry, type or status in the persistence. This exception is thrown by the ExpenseEntryManager, ExpenseEntryTypeManager and ExpenseEntryStatusManager.

NullPointerException
This exception is thrown in various methods where null value is not acceptable.
Refer to the documentation in Poseidon for more details. Not used in any of the new methods, but used in the old ones or the modified ones.

IllegalArgumentException
This exception is thrown in various methods if the given string argument is empty. Refer to the documentation in Poseidon for more details. The new methods use it for null arguments too. It is also used for arrays with null elements and empty strings. Empty string definition varies among the methods (in some case all space is considered invalid argument, in other it is not). For example field names cannot be all spaces, but the "contains" value for the "like" search filter can be one or more spaces. The Poseidon documentation must be consulted to see the individual details of each method.

1.6 Thread Safety

This component is not thread safe. Thread safety was not a requirement. It is not thread safe because, for example, a user may request an entry while another may delete it at the same time. In order to achieve thread safety all the methods from the persistence layer have to be synchronized. I think that it is better not to add an overhead to this component by synchronizing all the methods from the persistence and let the application handle thread safety.

The new classes and methods do not change the thread safety. The new data classes (BasicInfo and ExpenseEntryRejectReason) are mutable so they are not thread safe. However, they will not be typically shared between threads. The user should be aware that in a concurrent environment, the DB data may be changed by other thread, making the data contained in a data object invalid (for example it can be deleted entirely by another thread).

The ExpenseEntry class has some new methods to modify the new list attribute. They are not thread safe. It can be made so by synchronizing on the list. Because of the nature of the class, it is not worth doing it though.

The ExpenseEntryManager class itself is thread safe because it is immutable. However, it relies on the ExpenseEntryPersistence implementations which are not thread safe. That makes the manager methods that delegate the calls to the persistence methods, not thread safe. Solutions are synchronizing everything (at the manager level or persistence level). A solution based on transactions can also be devised. However, either solution doesn't protect us from logical errors (one thread deletes an expense entry used by another thread).

The ExpenseEntryDbPersistence class is almost thread safe itself (it has a setConnection method that might produce strange results; that can be fixed by synchronizing all connection using methods). However, because the underlying storage is a database, the class is not thread safe anyway. To make it thread safe, synchronization on all methods or transactions (with the appropriate transaction isolation

flags to prevent dirty reads, etc) can be used. However, that does not prevent logical errors (one thread deletes an expense entry used by another thread).

The Criteria hierarchy is immutable so it is thread safe.

In conclusion, the component is not thread safe (not required) but it can be made so using synchronization and/or SQL transactions.

2. Environment Requirements

2.1 Environment

- Development language: Java 1.4
- Compile target: Java 1.3, Java 1.4

2.2 TopCoder Software Components

- **Configuration Manager 2.1.3 –** used to retrieve the configured data. This component is used by getting its singleton instance with ConfigManager.getInstance(). Then the existsNamespace method should be used to determine whether the namespace is already loaded. If not, the add method is used to load the default configuration file. Finally, getString returns the values of the properties. Version 2.1.4 is available. If the PM approves, it could be used instead of 2.1.3. It allows spaces in directory and file names and it doesn't depend on Xerces so using it seems a good choice.
- **GUID Generator 1.0** is used to assign unique ids to records. This component has the advantage of not requiring persistent storage (such as ID Generator requires), making the component easier to use. A generator is obtained with UUIDUtility.getGenerator(UUIDType.TYPEINT32). Then using generator.getNextUUID().toString() ids are generated as needed.
- **Base Exception 1.0** is used as a base class for the all the custom exceptions defined in this component. The purpose of this component is to provide a consistent way to handle the cause exception for both JDK 1.3 and JDK 1.4.
- **DB Connection Factory 1.0** provides a simple but flexible framework allowing the clients to obtain the connections to a SQL database without providing any implementation details. This component is used to obtain a connection to a database.

2.3 Third Party Components

None.

3. Installation and Configuration

3.1 Package Name

com.topcoder.timetracker.entry.expense

3.2 Configuration Parameters

Parameter	Description	Values
connection_producer_name	Identifies a ConnectionProducer which will be used to obtain a connection to a database.**Optional**	Expense_Entry _Connection _Producer
entry_persistence_class	Fully qualified class name of the ExpenseEntryPersistence implementation. **Required**.	ExpenseEntryDbPersistence
entry_type_persistence_class	Fully qualified class name of the ExpenseEntryTypePersistence implementation. **Required**.	ExpenseEntryTypeDbPersistence
entry_status_persistence_class	Fully qualified class name of the ExpenseEntryStatusPersistence implementation. **Required**.	ExpnseEntryStatusDbPersistence

Here is an example of the configuration file:

```
<?xml version="1.0" ?>
  <CMConfig>
     <Config name=" com.topcoder.timetracker.entry.expense">
           <!--the  class name of the entry persistence-->
           <property name=" entry_persistence_class">
              <value>ExpenseEntryDbPersistence</value>
           </property>
           <!--the  class name of the entry type persistence-->
           <property name=" entry_type_persistence_class">
              <value>ExpenseEntryTypeDbPersistence</value>
           </property>
           <!--the  class name of the entry persistence-->
           <property name=" entry_status_persistence_class">
              <value>ExpenseEntryStatusDbPersistence</value>
           </property>
           <!--the  name identifying a ConnectionProducer instance-->
           <property name=" connection_producer_name">
              <value>Expense_Entry_Connection_Producer</value>
           </property>
 </Config>
```

3.3 Dependencies Configuration

None.

4. Usage Notes

4.1 Required steps to test the component

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

4.2 Required steps to use the component

Extract the component distribution.

4.3 Demo

For this demo I will use the configuration file shown above.

```
//create an ExpenseEntryType
ExpenseEntryType type1 = new ExpenseEntryType();
//set the fields of type1
type1.setDescription("Air Transportation");
type1.setCreationUser("John");
type1.setModificationUser("Tom");

//create another ExpenseEntryType
//this time supply the id
//if the id exist in the ExpenseTypes table then type2 will not be added to the table
ExpenseEntryType type2 = new ExpenseEntryType(2);
//set the fields of type2
type2.setDescription("Car Rental");
type2.setCreationUser("Jimmy");
type2.setModificationUser("Aristotel");

//create an ExpenseEntryTypeManager
ExpenseEntryTypeManager typeManager = new
ExpenseEntryTypeManager("com.topcoder.timetracker.entry.expense");


//add the types to the ExpenseTypes table
typeManager.addType(type1);
typeManager.addType(type2);
//assume that both types have been added to the ExpenseTypes table

//create an ExpenseEntryStatus
ExpenseEntryStatus status1 = new ExpenseEntryStatus();
//set the fields of the status1
status1.setDescription("Approved");
status1.setCreationUser("John");
status1.setModificationUser("Tom");

//create another ExpenseEntryStatus
//this time supply the id
//if the id exist in the ExpenseStatuses table then status2 will not be added to the
//table
ExpenseEntryStatus status2 = new ExpenseEntryStatus(3);
//set the fields of the status1
status2.setDescription("Not Approved");
status2.setCreationUser("Mike");
status2.setModificationUser("Tom");

//create an ExpenseEntryStatusManager
ExpenseEntryStatusManager statusManager = new
ExpenseEntryStatusManager("com.topcoder.timetracker.entry.expense");

//add the statuses to the ExpenseStatuses table
statusManager.addStatus(status1);
statusManager.addStatus(status2);
//assume that both statuses have been added to the ExpenseStatuses table

//create an ExpenseEntry
```

```
ExpenseEntry entry1 = new ExpenseEntry();
//set the fields of entry1
entry1.setDescription("project Ohio");
entry1.setCreationUser("George");
entry1.setModificationUser("George);
entry1.setDate(200000);
entry1.setExpenseType(type1);
entry1.setExpenseStatus(status1);
entry1.setBillable(true);
double amount =20000;
entry1.setAmount(amount);

//add reject reasons (by id), one at a time, or several at a time
entry1.addRejectReason(23);
entry1.addRejectReason(24);
entry1.addRejectReasons(new int[] {25, 26, 27});

//create another ExpenseEntry
//this time supply the id
//if the id exist in the ExpenseEntries table then entry2 will not be added to the
//table
ExpenseEntry entry2 = new ExpenseEntry(3);
//set the fields of entry2
entry2.setDescription("project New York");
entry2.setCreationUser("Alonso");
entry2.setModificationUser("Alonso);
entry2.setDate(200000);
entry2.setExpenseType(type2);
entry2.setExpenseStatus(status2);
entry2.setBillable(false);
amount =200000;
entry2.setAmount(amount);

//create an ExpenseEntryManager
ExpenseEntryManager entryManager = new
ExpenseEntryManager("com.topcoder.timetracker.entry.expense");

//add the entries to the ExpenseEntries table
//for entry1, the reject reasons added above are persisted too
entryManager.addEntry(entry1);
entryManager.addEntry(entry2);
//assume that both entries have been added to the ExpenseEntries table

//retrieve type(s), status(es) and entry(entries) from the database
ExpenseEntryType type3 = typeManager.retrieveType(2);
System.out.println("description = "+type3.getDescription());
ExpenseEntryStatus status3 = statusManager.retrieveStatus(2);
System.out.println("creation user = "+status2.getCreationUser());
ExpenseEntry entry3 = entryManager.retrieveEntry(3);
System.out.println("is billable = "+entry3.isBillable());

List entries = entryManager.retrieveAllEntries();
for(int i=0;i<entries.size();i++){
   ExpenseEntry e = (ExpenseEntry)entries.get(i);
   System.out.println("is billable+"e.isBillable());
```

```
    //get the reject reasons and print them
    ExpenseEntryRejectReason[] r = e.getRejectReasons();
    for (int j = 0; j < r.length; j++) {
        System.out.println(r[j].getId() + " " + r[j].getCreationUser() + " "
                + r[j].getCreationDate());
    }

    //get the reject reason ids directly and print them
    int[] rids = e.getRejectReasonIds();
    for (int j = 0; j < rids.length; j++) {
        System.out.println(rids[j].getId());
    }
}

List types = typeManager.retrieveAllTypes();
for(int i=0;i<types.size();i++){
    ExpenseEntryType t = (ExpenseEntryType)types.get(i);
    System.out.println("description+"t.getDescription());
}

List statuses = statusManager.retrieveAllStatuses();
for(int i=0;i<statuses.size();i++){
    ExpenseEntryStatus s = (ExpenseEntryStatus)statuses.get(i);
    System.out.println("description+"t.isBillable());
}

//update an entry
entry3.setBillabel(true);

//update the entry by adding a new reject reason
entry3.addRejectReason(50);

//and deleting old ones (one or several at a time)
entry3.deleteRejectReason(51);
entry3.deleteRejectReason(52);
entry3.deleteRejectReasons(new int[] {53, 54, 55});

//the existing reject reasons can also be changed
//the code below takes each reject reason and changes
//the modification user
ExpenseEntryRejectReason[] r = entry3.getRejectReasons();
for (int i = 0; i < r.length; i++) {
    r[i].setModificationUser("me");
    entry3.updateRejectReason(r[i]);
}

//alternatively the update can be done for all reject reasons at a time
entry3.updateRejectReasons(r);

//the entry update method will make sure the reject reasons from the
//database are in synch with the entry reject reasons (that means records
//could be added, deleted or updated)
entryManager.updateEntry(entry3);

//update a type
type3.setModificationUser("Pam);
```

```
typeManager.updateType(type3);

//update a status
status3.setDescription(“Pending Approval”);
statusManager.updateStatus(status3);

//delete an entry
if (entryManager.deleteEntry(3))
   System.out.println(“entry was deleted”);

//delete all entries
entryManager.deleteAllEntries();

//delete a type
if (typeManager.deleteType(2))
   System.out.println(“type was deleted”);

//delete all types
typeManager.deleteAllTypes();

//delete a status
if (statusManager.deleteStatus(3))
   System.out.println(“status was deleted”);

//delete all statuses
statusManager.deleteAllStatuses();

//the alternative to the addEntry, deleteEntry, updateEntry and retrieveEntry are
//the BULK methods that can process more entries in one call

//three entries are added in one call, atomically (meaning if one fails, all fail
//without any database changes)
entryManager.addEntries(new ExpenseEntry[] {entry1, entry2, entry3}, true);

//three entries are added in one call, non atomically (meaning if one fails, the
//others are still performed independently and the failed ones are returned to
//the user)
ExpenseEntry[] failed =
       entryManager.addEntries(new ExpenseEntry[] {entry1, entry2, entry3}, false);
for (int i = 0; i < failed.length; i++) {
      System.out.println(failed[i].getDescription() + “ adding failed”);
}

//three entries are deleted in one call, atomically (meaning if one fails, all fail
//without any database changes)
entryManager.deleteEntries(new int[] {50, 51, 52}, true);

//three entries are deleted in one call, non atomically (meaning if one fails, the
//others are still performed independently and the ids of the failed ones are
//returned to the user)
int[] failedIds =
       entryManager.addEntries deleteEntries(new int[] {50, 51, 52}, false);
for (int i = 0; i < failedIds.length; i++) {
      System.out.println(failedIds[i] + “ deletion failed”);
}
```

```
//three entries are updated in one call, atomically (meaning if one fails, all fail
//without any database changes)
entryManager.updateEntries(new ExpenseEntry[] {entry1, entry2, entry3}, true);

//three entries are updated in one call, non atomically (meaning if one fails, the
//others are still performed independently and the failed ones are returned to
//the user)
ExpenseEntry[] failed =
        entryManager.updateEntries(new ExpenseEntry[] {entry1, entry2, entry3}, false);
for (int i = 0; i < failed.length; i++) {
        System.out.println(failed[i].getDescription() + " update failed");
}

//three entries are retrieved in one call, atomically (meaning if one fails, all fail
//without any results being returned)
ExpenseEntry[] entries = entryManager.retrieveEntries(new int[] {50, 51, 52}, true);
if (entries == null) {
        System.out.println("retrieval failed");
} else for (int i = 0; i < entries.length; i++) {
        System.out.println(entries[i].getDescription() + " retrieved");
}

//three entries are retrieved in one call, non atomically (meaning if one fails, the
//others are still retrieved independently and the retrieved ones are returned to
//the user – note the difference from add/delete/update)
ExpenseEntry[] entries =
        entryManager.retrieveEntries(new int[] {50, 51, 52}, false);
for (int i = 0; i < entries.length; i++) {
        System.out.println(entries[i].getDescription() + " retrieved");
}

//the SEARCH framework allows expense entries being searched based
//on different criteria

//look for description containing a given string
Criteria crit1 = FieldLikeCriteria.getDescriptionContainsCriteria("gambling debt");

//look for expense status, expense type, billable flag, creation and modification
users
// matching a given value
Criteria crit2 = FieldMatchCriteria.getExpenseStatusMatchCriteria(2);
Criteria crit3 = FieldMatchCriteria.getExpenseTypeMatchCriteria(23);
Criteria crit4 = FieldMatchCriteria.getBillableMatchCriteria(true);
Criteria crit5 = FieldMatchCriteria.getCreationUserMatchCriteria("me");
Criteria crit6 = FieldMatchCriteria.getModificationUserMatchCriteria("boss");

//look for amount between two given value
//the null calls means open ended (the first means amount >= 1000, the second
amount <=2000)
Criteria crit7 = FieldBetweenCriteria.getAmountBetweenCriteria(
            BigDecimal.valueOf(1000), BigDecimal.valueOf(2000));
Criteria crit8 = FieldBetweenCriteria.getAmountBetweenCriteria(
            BigDecimal.valueOf(1000), null);
Criteria crit9 = FieldBetweenCriteria.getAmountBetweenCriteria(
            null, BigDecimal.valueOf(2000));
```

```
//look for creation and modification dates between two given dates
//the null calls mean open ended (the first means 2005/30/1 or later, the second today or before)
Criteria crit10 = FieldBetweenCriteria.getCreationDateBetweenCriteria(
        new Date(2005, 30, 1), new Date());
Criteria crit11 = FieldBetweenCriteria.getCreationDateBetweenCriteria(
        new Date(2005, 30, 1), null);
Criteria crit12 = FieldBetweenCriteria.getCreationDateBetweenCriteria(
        null, new Date());
Criteria crit13 = FieldBetweenCriteria.getModificationDateBetweenCriteria(
        new Date(2005, 30, 1), new Date());
Criteria crit14 = FieldBetweenCriteria.getModificationDateBetweenCriteria(
        new Date(2005, 30, 1), null);
Criteria crit15 = FieldBetweenCriteria.getModificationDateBetweenCriteria(
        null, new Date());

//look for an expense entry having a given reject reason
Criteria crit16 = new RejectReasonCriteria(50);
Criteria crit16b = new RejectReasonCriteria(51);
Criteria crit16c = new RejectReasonCriteria(52);

//look for entries not matching a given criteria
Criteria crit17 = new NotCriteria(crit10);

//look for entries matching two criteria at the same time
Criteria crit18 = CompositeCriteria.getAndCompositeCriteria(crit2, crit6);

//look for entries matching any of two criteria
//in this particular case it looks for an entry having reject reason 51 OR 52
Criteria crit19 = CompositeCriteria.getOrCompositeCriteria(crit16b, crit16c);

//look for entries matching more criteria at the same time
//in this particular case it looks for an entry having reject reason 50, 51 AND 52
Criteria crit20 = CompositeCriteria.getAndCompositeCriteria(
        new Criteria[] {crit16, crit16b, crit16c});

//look for entries matching any of more criteria
Criteria criteria = CompositeCriteria.getOrCompositeCriteria(
        new Criteria[] {crit2, crit6, crit12});

// the actual search and result printing
ExpenseEntry[] entries = entryManager.searchEntries (criteria);
for (int i = 0; i < entries.length; i++) {
      System.out.println(entries[i].getDescription() + " match found");
}
```

5. Future Enhancements

More ExpenseEntryPersistence implementations to this component.

The Criteria interface allows additional search criteria to be used without any code changes to the component. The first enhancement one can think of is adding more implementations.