Expense Entry 3.2 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, ExpenseType and ExpenseStatus.

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)

Version 2.0 adds a few additional functionalities:

- Table name and fields have been changed.
- Company entity added
- For Expense Entry/Type, one company id is associated with. Each expense entry/type should be associated with one company id, while company id can map to many expense entry/type. Such mapping is guaranteed by this component.
- Filters: And company id filter for expense entry/type, and filters for expense status is added.

Version 3.1 adds some changes

- Invoice has been added to the Expense Entry
- All entries are now JavaBeans and follow the specific guidelines for JavaBeans, this is accomplished by extending the entities in this component from BaseEntry and from TimeTrackerEntry java beans.
- New Search Criteria has been added to search for Expense Entries based on Invoice id
- ExpenseEntry actions (that modify data) have now the ability audit the changes (via the Time tracker Audit 3.1 component)
- Reject Reason Functionality has now been extended from BaseEntry.

New features in version 3.2:

- Introduces a session bean to facilitate transactional control for each entity. This was a major issue with the 3.1 version. As a result, the DAO implementation documentation as well as the relevant SDs is modified to remove all manual transaction control. In fact, to ease the transition, the quantity of SD has been paired down to a manageable size.
- The v3.1 managers will become delegates to these beans. As such, the managers will be split into an interface with their original names name, and a delegate implementation for EJB integration. Entity validation is put in the bean, and ID generation is moved to the DAO.
- Only local access to the bean is needed.
- Transactional control uses the "Required" and "RequiresNew" levels to guarantee that any action is always under such control.
- Updated documentation and presentation for most classes, even outside the scope of v3.2 requirements. The persistence classes have been renamed.

DESIGN CONSIDERATIONS:

Non-atomic operations and transactional control

The requirement calls for keeping the non-atomic batch operations while incorporating them into the container-managed transaction demarcation. Because non-atomic operations in their v3.1 version require that they be able to roll back individual entities, this was simply not possible. The suggestion was to use with them a transaction level of “”NotSupported” and simply let them control themselves manually. This designer rejected this as being error-prone (as most such manual transactions are) and not elegant. Instead, a solution is utilized where another EJB is made available via deployment that performs individual operations as dedicated transactions. This allows a bean that needs to perform non-atomic batch operations to simply delegate each entity to that EJB and allow the container to still control each such call in a separate transaction that can be individually rolled back without affecting the current transaction this bean is in. For example, the ExpenseEntryBean’s addEntries method, when called with non-atomic flag, will call the addEntry method in BasicExpenseEntryBean for each passed ExpenseEntry. The result is that there is no messy manual transaction control.

Incidentally, to facilitate this design choice, the design splits the ExpenseEntry EJB into its basic component operations (non-batch) and full component operations (non-batch and batch). This allows for the basic EJB to be elegantly reused as a dedicated transactional EJB. Also, to enhance the component usage, each batch operation is overloaded so there is a default operation that is atomic. This allows implementations to easily skip non-atomic batch operations by simply not implementing the full-parameter batch methods.

The Sorry State of v3.1 Documentation

Any reader of the v3.1 component will quickly discover how mangled, inconsistent, and archaic the whole presentation is. It frequently makes reference to ExpenseEntryType and ExpenseType, yet they refer to the same entity. The Persistence objects made use of connection setters, which in the context of required transactional control is not useful. Therefore, a great deal of the design has been brought up to current TopCoder standards, save for any SQL operations, which appear to remain correct, and Sequence Diagrams.

As part of this process, all persistence interfaces and implementations have been renamed in accordance with recent acquired conventions in v3.2 designs.

Furthermore, all implementations now make use of a connection factory that is created via ObjectFactory. The naming of expense types and statuses’ classes has been standardized.

I have also reverted to the use of PersistenceException in the managers as the v3.1 addition of ExpenseEntryManagerException seemed artificial, provided little value, and was inconsistently applied.

As noted above, I have removed most duplicate SDs as they really provided little value, but I introduced new ones that pertain to the instantiation of the delegate and initialization of the EJB, as well as an overview of the flow of calls from the application to the DAO via the delegate and EJB.

EJBs and ConfigManager:

The EJB specification stipulates that File I/O is not allowed during the execution of the bean. At first glance this might mean that the use of the Config Manager, and by extension Object Factory, DB Connection Factory, and ID generator, could not be allowed because it performs property retrieval and storing using files. However, the restriction is only placed on the bean’s lifetime, not the ConfigManager’s. Therefore, as long as the Config Manager does not perform I/O itself during the execution of the bean, or to be more accurately, that the thread performing a bean operation does not cause the ConfigManager to perform I/O in the course of the bean's execution, the use of Config Manager to hold an in-memory library of properties is acceptable. Therefore, this design makes extensive use of Config Manager, Object Factory, DB Connection Factory, and ID generator, again, with the stipulation that the Config Manager implementation does not perform I/O when this component is retrieving properties. The current default implementation of the Config Manager holds all loaded properties in a map, so all property reading is done in-memory.

In general, and aside to the central issue here, the purpose behind the EJB specification stipulation against I/O during the business operation of the bean exists to accommodate portability. In practice, since portability is not always a reason behind the choice to use EJBs, many containers allow direct I/O in the bean. In fact, since the main purpose of the use of EJBs in this application is to enforce transactional control, I/O may be allowed as long as the access can be transitionally controlled, something that is a challenge in itself.

Serialization and Filter:

The Filter hierarchy in Search Builder is currently not serializable, which means it currently cannot be used. The PM is currently researching solutions to this.

JNDI and DBConnectionFactory:

The TxDataSource required for this component can still be obtained as a javax.sql.DataSource. As such, the JNDIConnectionProvider available in the DBConnection Factory component can still perform connection creation services for this design. As such, it is kept unaltered in the DAO implementations.

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, ExpenseType, and ExpenseStatus 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.

Business Delegate Pattern is used by manager implementations so the user is decoupled from the intricacies of obtaining and calling the session EJBs.

Data Access Object Pattern is used in the DAO classes.

1.2 Industry Standards

SQL, JDBC
JavaBeans (http://java.sun.com/products/javabeans/docs/spec.html)
EJB 2.1

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.

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.expense_entry_id = expense_entry. expense_entry_id)".
```

Other ways can be used as well:

```
- EXISTS (SELECT reject_reason_id FROM exp_reject_reason "
  + "WHERE exp_reject_reason. expense_entry_id = expense_entry. expense_entry_id"
  + "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.

1.3.2 *Auditing*

The Audit component requires the consumer to identify the application area that the audit is for. The application area for the Expense Entry will be TT_EXPENSE. Thus in the AuditHeader we will need to set this accordingly. Here is what we need to do:

1. We create a new AuditHeader instance and we initialize it as follows:
    a. We set the applicationArea to be ApplicationArea.TT_EXPENSE
    b. We set the entityId to the id of the entity that we are auditing
    c. We also need to set the table name to “expense_entry:
    d. Set other properties if appropriate (such as user name, company id, client id, project id, etc…) in most cases we leav this blank.
2. We need to create AuditDetail instance for each field in the table/bean that is changing, this is done as follows:
    a. Set the name of the field we are changing (such as “amount” for example)
    b. Set the old value
    c. Set the new value

Once this is done we simply execute the AuditManager.createAuditRecord(auditRecord);

1.3.3 Transaction Control

Transaction control is managed by the EJB Container. No entity in this component will manage them manually. Transactional control is now done declaratively in the deployment descriptor. All rollbacks are done in the session beans. All of this should be standard knowledge to experienced EJB developers.

1.4 Component Class Overview

1.4.1 com.topcoder.timetracker.entry.expense

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.

Version 2.0 adds a company id to associated with it. In current usage, one expense entry should and only map to one company id.

Version 3.1 Adds the ability to deal with Invoices. The entry is now extended from BaseEntry javabean

A member is also added for the mileage involved (if it is an Auto Mileage expense).

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.

ExpenseType

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 ExpenseType 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.
Version 2.0 add active and company id to associated with it. In current usage, one expense entry type should and only map to one company id. And has an active status.
Version 3.1 changes the way the entry is extended. The entry is now extended from TimeTrackerEntry javabean.

ExpenseStatus
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 ExpenseStatus 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.
Version 3.1 changes the way the entry is extended. The entry is now extended from TimeTrackerEntry javabean.

ExpenseTypeManager
This interface defines the contract for the complete management of an expense type. It provides CRUDE and extensive search operations. It has one implementation in this design: ExpenseTypeManagerLocalDelegate.

ExpenseStatusManager
This interface defines the contract for the complete management of an expense status. It provides CRUDE and extensive search operations. It has one implementation in this design: ExpenseStatusManagerLocalDelegate.

ExpenseEntryManager
This interface defines the contract for the complete management of an expense entry. It provides single and batch CRUD operations with the choice to perform audits on all writeable operations. Furthermore, all batch operations support selective operations, so these calls can be either atomic or non-atomic. Atomic mode means that a failure on one entry causes the entire operation to fail. Non-atomic means that a failure in one entry doesn't affect the other and the user has a way to know which ones failed. It also supports robust searches. It has one implementation in this design: ExpenseEntryManagerLocalDelegate.

1.4.2 *com.topcoder.timetracker.entry.expense.criteria*

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 13 constants and 13 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 3 constants and 3 static methods 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 7 constants and 7 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.

1.4.3 com.topcoder.timetracker.entry.expense.persistence

ExpenseEntryDAO (formerly ExpenseEntryPersistence)
This interface defines the contract for the complete persistence management of an expense entry. It provides single and batch CRUD operations with the choice to perform audits on all writeable operations. Furthermore, all batch operations support selective operations, so these calls can be either atomic or non-atomic. Atomic mode means that a failure on one entry causes the entire operation to fail. Non-atomic means that a failure in one entry doesn't affect the other and the user has a way to know which ones failed. It also supports robust searches. It has one implementation in this design: InformixExpenseEntryDAO.

InformixExpenseEntryDAO (formerly ExpenseEntryDbPersistence)
This class is a concrete implementation of the ExpenseEntryDAO interface that uses an Informix database as the data store. This implementation uses the DB Connection Factory component to obtain a connection to the database, the GUID Generator component to create IDs for new Expense Entries. This class provides two constructors that will use Config Manager and Object Factory components to obtain instances of the connection factory and audit manager. One constructor will take a namespace, and the second will work with a default namespace.

This implementation is tailored to work in an EJB container and will rely on it to manage all transactions. To this end, it does not implement any of the batch operations that give the caller the choice to select whether the operations are atomic or not. In the case of the EJBs that come with this component, these will perform non-atomic operations by using the single entity methods.

As mentioned, the caller will have the chance to audit all writeable operations using the Audit Manager.

ExpenseTypeDAO (formerly ExpenseTypePersistence)
This interface defines the contract for the complete management of an expense entry type. It provides CRUDE methods for this purpose. It additionally provides robust search capabilities. It has one implementation in this design: InformixExpenseTypeDAO.

InformixExpenseTypeDAO (formerly ExpenseTypeDbPersistence)
This class is a concrete implementation of the ExpenseTypeDAO interface that uses an Informix database as the data store. This implementation uses the DB Connection

Factory component to obtain a connection to the database, the GUID Generator component to create IDs for new Expense Entry Types. This class provides two constructors that will use Config Manager and Object Factory components to obtain instances of the connection factory. One constructor will take a namespace, and the second will work with a default namespace. All methods are implemented.

ExpenseStatusDAO (formerly ExpenseTypePersistence)
This interface defines the contract for the complete management of an expense entry status. It provides CRUDE methods for this purpose. It additionally provides robust search capabilities. It has one implementation in this design: InformixExpenseStatusDAO.

InformixExpenseStatusDAO (formerly ExpenseTypeDbPersistence)
This class is a concrete implementation of the ExpenseStatusDAO interface that uses an Informix database as the data store. This implementation uses the DB Connection Factory component to obtain a connection to the database, the GUID Generator component to create IDs for new Expense Entry Statuses. This class provides two constructors that will use Config Manager and Object Factory components to obtain instances of the connection factory. One constructor will take a namespace, and the second will work with a default namespace. All methods are implemented.

1.4.4 com.topcoder.timetracker.entry.expense.ejb

ExpenseEntryManagerLocalDelegate
Implements the ExpenseEntryManager interface to provide management of the expense entries through the use of a local session EBJ - ExpenseEntryLocal. It will obtain the handle to the bean's local interface and will simply delegate all calls to it. It implements all methods.

BasicExpenseEntryLocalHome
The local home interface of the Basic Expense Entry EJB. It contains one, no-argument create method to create an instance of the local Basic Expense Entry.

BasicExpenseEntryLocal
The local interface of the Basic Expense Entry EJB, which provides access to the persistent store for expense entries managed by the application. It provides the basic, non-batch operations for managing an expense entry. BasicExpenseEntryBean is the corresponding bean that will perform the actual tasks.

BasicExpenseEntryBean
The session EJB that handles the actual manager requests for basic expense entry operations, as defined by the corresponding local interface – BasicExpenseEntryLocal. It simply delegates all operations to the ExpenseEntryDAO instance it obtains from the ObjectFactory.

ExpenseEntryLocalHome
The local home interface of the Expense Entry EJB. It contains one, no-argument create method to create an instance of the local Expense Entry.

ExpenseEntryLocal
The local interface of the Expense Entry EJB, which provides access to the persistent store for expense entries managed by the application. It extends BasicExpenseEntryLocal by providing batch operations. ExpenseEntryBean is the corresponding bean that will perform the actual tasks.

ExpenseEntryBean
The session EJB that handles the actual manager requests for expense entry operations, as defined by the corresponding local interface – ExpenseEntryLocal. As such, it extends BasicExpenseEntryBean and implements the batch methods, but with a wrinkle. All atomic batch operations simply delegate all operations to the ExpenseEntryDAO instance it obtains from the ObjectFactory, but all non-atomic calls are handled via the configured basic ExpenseEntry EJB, so the later can perform each operation atomically without involving the transaction that this instance is part of. For the duration of the calls, the local transaction is suspended and does not take part in the calls to the basic ExpenseEntry EJB.

ExpenseTypeManagerLocalDelegate
Implements the ExpenseTypeManager interface to provide management of the expense types through the use of a local session EBJ - ExpenseTypeLocal. It will obtain the handle to the bean's local interface and will simply delegate all calls to it. It implements all methods.

ExpenseTypeLocalHome
The local home interface of the Expense Type EJB. It contains one, no-argument create method to create an instance of the local Expense Type.

ExpenseTypeLocal
The local interface of the Expense Type EJB, which provides access to the persistent store for expense types managed by the application. It provides the operations for managing an expense type. ExpenseTypeBean is the corresponding bean that will perform the actual tasks.

ExpenseTypeBean
The session EJB that handles the actual manager requests for expense type operations, as defined by the corresponding local interface – ExpenseTypeLocal. It simply delegates all operations to the ExpenseTypeDAO instance it obtains from the ObjectFactory.

ExpenseStatusManagerLocalDelegate
Implements the ExpenseStatusManager interface to provide management of the expense statuses through the use of a local session EBJ - ExpenseStatusLocal. It will obtain the handle to the bean's local interface and will simply delegate all calls to it. It implements all methods.

ExpenseStatusLocalHome
The local home interface of the Expense Status EJB. It contains one, no-argument create method to create an instance of the local Expense Status.

ExpenseStatusLocal
The local interface of the Expense Status EJB, which provides access to the persistent store for expense statuses managed by the application. It provides the operations for managing an expense status. ExpenseStatusBean is the corresponding bean that will perform the actual tasks.

ExpenseStatusBean
The session EJB that handles the actual manager requests for expense status operations, as defined by the corresponding local interface – ExpenseStatusLocal. It simply delegates all operations to the ExpenseStatusDAO instance it obtains from the ObjectFactory.

1.5 Component Exception Definitions

PersistenceException[custom]
The PersistenceException exception is used to wrap any persistence implementation specific exception. These exceptions are thrown by the persistence interfaces and implementations, EJBs, and the manager interfaces and 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 (EJB) and delegate layer (the manager classes) will forward them to the user.

ConfigurationException[custom]
This exception is thrown by the manager and DAO implementations 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 persistence interfaces and implementations, EJBs, and the manager interfaces and implementations.

IllegalArgumentException
This exception is thrown in various methods if the given argument is null or 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 almost thread-safe. ExpenseEntry, ExpenseType, and ExpenseStatus are mutable and not thread-safe, but it is not expected that an instance of these will be manipulated by more than one thread at a time. All other classes are immutable and thus thread-safe.

With the introduction of the EJB container, we are in fact assured that each EJB, and by extension, each DAO implementation, will be accessed by no more than one thread at a time. The EJB container provides this kind of thread-safety.

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 2.1.5 –** used to retrieve the configured data.
- **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.
- **Time Tracker audit 3**.2 provides the functionality needed for auditing.
- **Time Tracker Invoice 3**.2 provides the functionality for invoice persistence
- **Time Tracker Reject Reason 3.2** for reject Reason capabilities
- **Time Tracker Common 3.2** provides the TimeTrackerEntry bean
- **Time Tracker Base Entry 3.2** provides the BaseEntry bean
- **Object Factory 2.0.1** provides the functionality for creating the objects dynamically
- **JNDI Utility 1.0** provides abstracted Context access for JNDI access.
- **Type Enum 1.0** provides the enum classes.

2.3 Third Party Components

None.

3. Installation and Configuration

3.1 Package Name

com.topcoder.timetracker.entry.expense
com.topcoder.timetracker.entry.expense.persistence
com.topcoder.timetracker.entry.expense.criteria
com.topcoder.timetracker.entry.expense.ejb

3.2 Configuration Parameters

3.2.1 Delegates and EJBs

Property Name	Description	Details	Required
jndi_reference	JNDI reference to the local contact EJB.	Example: “java:comp/env/ejb/ContactLocal”	Yes
jndi_context	JNDI context name, to be used with JNDIUtils	Example: “myContext”	No

3.2.2 DAOs

Property Name	Description	Details	Required
specification_namespace	The namespace used to create CM Specification Factory	String	Yes
connection_name	The name used to get connection from connection factory	String	No
db_connection_factory_key	Key for the DBConnectionFactory instance to pass to object factory	String	Yes

audit_manager_key	Key for the AuditManager instance to pass to object factory	String	Yes
type_manager_key	Key for the ExpenseTypeManager instance to pass to object factory	String	Yes
status_manager_key	Key for the ExpenseStatusManager instance to pass to object factory	String	Yes
invoice_manager_key	Key for the InvoiceManager instance to pass to object factory	String	Yes
rejectreason_dao_key	Key for the RejectReasonDAO instance to pass to object factory	String	Yes

3.2.3 Sample Configurations

Please refer to j2ee_config.xml and the sample deployment descriptor ejb-jar.xml. All are in the docs/examples directory.

3.3 Dependencies Configuration

The dependency components should be configured according to their documentation.

The EJB deployment descriptor must have a data source configured for the DBConnection Factory. Also, it must include the following two environment entries for the all beans:

Parameter	Description	Details
SpecificationNamespace	Namespace to use with the ConfigManagerSpecificationFactory Required	Example: “com.topcoder.specification”
ExpenseEntryDAOKey ExpenseStatusDAOKey ExpenseTypeDAOKey	Key for the ExpenseEntryDAO/ ExpenseStatusDAO/ExpenseTypeDAO instance to pass to object factory Required.	“expenseEntryDAO” “expenseStatusDAO” “expenseTypeDAO”

For the ExpenseEntryBean, it must also provide an EJB reference for “ejb/BasicExpenseEntry” that provides this bean with a reference to the single operations it needs to perform non-atomic batch operations.

The sample deployment descriptor introduced in 3.2.5 has a simple example of these

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, the example configuration and deployment files introduced in 3.2.3 can be used as a staring point.

4.3.1 Expense Status

// Demostrates manipulations on expense entry statuses and persistence.

```
        // Create a manager for expense entry status from configuration
        ExpenseStatusManager manager = new
ExpenseStatusManagerLocalDelegate();

        // Create an expense status with ID
        ExpenseStatus status = new ExpenseStatus(5);

        // Create an expense status without ID. The ID will be
generated when adding it to the persistence.
        status = new ExpenseStatus();

        // Set fields
        status.setDescription("Description");
        status.setCreationUser("Create");
        status.setModificationUser("Create");

        // Add the expense entry status to persistence.
        boolean success = manager.addStatus(status);

        assertTrue("The instance should be added to the persistence.",
success);

        // Change fields
        status.setModificationUser("Modify");
        status.setDescription("Changed");

        // Update the expense entry status to persistence
        success = manager.updateStatus(status);

        assertTrue("The instance should be updated.", success);

        // Get the expense entry status from persistence
        ExpenseStatus retrieved =
manager.retrieveStatus(status.getId());

        // Get properties
        long id = retrieved.getId();
        String description = retrieved.getDescription();
        String creationUser = retrieved.getCreationUser();
        String modificationUser = retrieved.getModificationUser();
        Date creationDate = retrieved.getCreationDate();
        Date modificationDate = retrieved.getModificationDate();

        UnitTestHelper.assertEquals(status, retrieved);

        // Retrieve a list of all expense entry statuses from
persistence
        ExpenseStatus[] list = manager.retrieveAllStatuses();

        // Delete one expense entry status from persistence
        success = manager.deleteStatus(status.getId());
        assertTrue("The expense entry status should be deleted.",
success);

        // Delete all expense entry statuses
              manager.deleteAllStatuses();
```

4.3.2 Expense Type

```
      // Demostrates manipulations on expense entry types and persistence.

       // Create a manager for expense entry type from configuration
        ExpenseTypeManager manager = new
ExpenseTypeManagerLocalDelegate();

        // Create an expense type with ID
        ExpenseType type = new ExpenseType(4);

        // Create an expense type without ID. The ID will be generated
when adding it to the persistence.
        type = new ExpenseType();

        // Set fields
        type.setDescription("Description");
        type.setCreationUser("Create");
        type.setModificationUser("Create");
        type.setCompanyId(1);

        // Add the expense entry entry to persistence.
        boolean success = manager.addType(type);

        assertTrue("The instance should be added to the persistence.",
success);

        // Change fields
        type.setModificationUser("Modify");
        type.setDescription("Changed");

        // Update the expense entry type to persistence
        success = manager.updateType(type);

        assertTrue("The instance should be updated.", success);

        // Get the expense entry type from persistence
        ExpenseType retrieved = manager.retrieveType(type.getId());

        // Get properties
        long id = retrieved.getId();
        String description = retrieved.getDescription();
        String creationUser = retrieved.getCreationUser();
        String modificationUser = retrieved.getModificationUser();
        Date creationDate = retrieved.getCreationDate();
        Date modificationDate = retrieved.getModificationDate();

        UnitTestHelper.assertEquals(type, retrieved);
        // Retrieve a list of all expense entry types from persistence
        ExpenseType[] list = manager.retrieveAllTypes();

        // Delete one expense entry type from persistence
        success = manager.deleteType(type.getId());

        assertTrue("The expense entry type should be deleted.",
success);

        // Delete all expense entry types
              manager.deleteAllTypes();
```

4.3.3 *Expense Entry*

// Demostrates manipulations on expense entries and persistence.

```
// Add one status and one type into database
        ExpenseStatusManager statusManager = new
ExpenseStatusManagerLocalDelegate();
        ExpenseTypeManager typeManager = new
ExpenseTypeManagerLocalDelegate();
        ExpenseStatus status = new ExpenseStatus(1);
        ExpenseType type = new ExpenseType(2);

        status.setCreationUser("Create");
        status.setDescription("Status");
        status.setModificationUser("Create");
        type.setCreationUser("Create");
        type.setDescription("Type");
        type.setModificationUser("Create");
        type.setCompanyId(1);

        statusManager.addStatus(status);
        typeManager.addType(type);

        // Create a manager for expense entry from configuration
        ExpenseEntryManager manager = new
ExpenseEntryManagerLocalDelegate();

        // Create an expense entry with ID
        ExpenseEntry entry = new ExpenseEntry(5);

        // Create an expense entry without ID. The ID will be generated
when adding it to the persistence.
        entry = new ExpenseEntry();
        entry.setCompanyId(1);

        // Set fields
        entry.setDescription("Description");
        entry.setCreationUser("Create");
        entry.setModificationUser("Create");
        entry.setAmount(new BigDecimal(100.00));
        entry.setBillable(true);
        entry.setDate(UnitTestHelper.createDate(2000, 1, 2));
        entry.setExpenseType(type);
        entry.setStatus(status);
        // Create the reject reason
        Map rejectReasons = new HashMap();
        RejectReason reason1 = new RejectReason();
        reason1.setId(1);
        reason1.setCreationDate(UnitTestHelper.createDate(2005, 1, 1));
        reason1.setModificationDate(UnitTestHelper.createDate(2005, 2,
1));
        reason1.setCreationUser("TangentZ");
        reason1.setModificationUser("Ivern");
        reason1.setDescription(description1);
        reason1.setCompanyId(1);
        rejectReasons.put(new Long(reason1.getId()), reason1);

        // add the reject reason to entry
        entry.setRejectReasons(rejectReasons);

        // Add the expense entry entry to persistence and do the audit
        boolean success = manager.addEntry(entry, true);

        assertTrue("The instance should be added to the persistence.",
success);

        // for the entry, the reject reasons added above are persisted
too
        ExpenseEntry[] entries = manager.retrieveAllEntries();

        for (int i = 0; i < entries.length; i++) {
            ExpenseEntry e = entries[i];
            System.out.println("is billable" + e.isBillable());

            //get the reject reasons and print them
            RejectReason[] r = (RejectReason[])
e.getRejectReasons().values().toArray(new RejectReason[0]);

            for (int j = 0; j < r.length; j++) {
                System.out.println(r[j].getId() + " " +
r[j].getCreationUser() + " " + r[j].getCreationDate());
            }
        }

        // Change fields
        entry.setModificationUser("Modify");
        entry.setDescription("Changed");
        entry.setAmount(new BigDecimal(10000.00));
        entry.setBillable(false);
        entry.setDate(UnitTestHelper.createDate(2005, 2, 1));

        // delete the reject reason
        rejectReasons.remove(new Long(reason1.getId()));
        entry.setRejectReasons(rejectReasons);
        reason1.setModificationUser("me");

        // Update the expense entry to persistence and do the audit
        success = manager.updateEntry(entry, true);

        assertTrue("The instance should be updated.", success);
        // Get the expense entry from persistence. The
        ExpenseEntry retrieved = manager.retrieveEntry(entry.getId());

        // Get properties
        long id = retrieved.getId();
        String description = retrieved.getDescription();
        String creationUser = retrieved.getCreationUser();
        String modificationUser = retrieved.getModificationUser();
        Date creationDate = retrieved.getCreationDate();
        Date modificationDate = retrieved.getModificationDate();
        Date date = retrieved.getDate();
        BigDecimal amount = retrieved.getAmount();
        boolean billable = retrieved.isBillable();

        UnitTestHelper.assertEquals(entry, retrieved);

        // Retrieve a list of all expense entries from persistence
        ExpenseEntry[] list = manager.retrieveAllEntries();

        assertEquals("One expense entry should be retrieved.", 1,
list.length);

        // Delete one expense entry from persistence and do the audit
        success = manager.deleteEntry(entry.getId(), true);

        assertTrue("The expense entry should be deleted.", success);

        // Delete all expense entries and do the audit
        manager.deleteAllEntries(true);

        // adding large block of entries.
        ExpenseEntry[] expected = new ExpenseEntry[5];

        // Add 5 instances
        for (int i = 0; i < 5; ++i) {
            expected[i] = new ExpenseEntry(i + 1);
            expected[i].setCompanyId(1);
            expected[i].setCreationUser("Create" + i);
            expected[i].setModificationUser("Modify" + i);
            expected[i].setDescription("Description" + i);
            expected[i].setAmount(new BigDecimal(i));
            expected[i].setBillable(true);
            expected[i].setDate(UnitTestHelper.createDate(2005, 2, i));
            expected[i].setExpenseType(type);
            expected[i].setStatus(status);
        }

        ExpenseEntry[] failed = manager.addEntries(expected, false,
false);

        for (int i = 0; i < failed.length; i++) {
            System.out.println(failed[i].getDescription() + " adding
failed");
        }

        // three entries are updated in one call, atomically (meaning
if one fails, all fail
        // without any database changes)
        manager.updateEntries(new ExpenseEntry[] {expected[0],
expected[1], expected[2]}, true, false);

        // 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)
        failed = manager.updateEntries(new ExpenseEntry[] {expected[0],
expected[1], expected[2]}, false, 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[] receive = manager.retrieveEntries(new long[] {1,
2, 3}, true);

        if (receive.length == 0) {
            System.out.println("retrieval failed");
        } else {
            for (int i = 0; i < receive.length; i++) {
                System.out.println(receive[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)
        receive = manager.retrieveEntries(new long[] {1, 2, 3}, false);

        for (int i = 0; i < receive.length; i++) {
            System.out.println(receive[i].getDescription() + "
retrieved");
        }

        // three entries are deleted in one call, atomically (meaning
if one fails, all fail
        // without any database changes)
        manager.deleteEntries(new long[] {1, 2, 3}, true, false);

        // 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)
        long[] failedIds = manager.deleteEntries(new long[] {1, 2, 3},
false, false);
```

```
        for (int i = 0; i < failedIds.length; i++) {
            System.out.println(failedIds[i] + " deletion failed");
        }
```

4.3.4 Search

// Demostrates manipulations on search framework.

```
  // Add one status and one type into database
        ExpenseStatusManager statusManager = new
ExpenseStatusManagerLocalDelegate();
        ExpenseTypeManager typeManager = new
ExpenseTypeManagerLocalDelegate();
        ExpenseStatus status = new ExpenseStatus(1);
        ExpenseType type = new ExpenseType(2);

        status.setCreationUser("Create");
        status.setDescription("Status");
        status.setModificationUser("Create");
        type.setCreationUser("Create");
        type.setDescription("Type");
        type.setModificationUser("Create");
        type.setCompanyId(1);

        statusManager.addStatus(status);
        typeManager.addType(type);

        // Create the reject reason
        Map rejectReasons = new HashMap();
        RejectReason reason1 = new RejectReason();
        reason1.setId(1);
        reason1.setCreationDate(UnitTestHelper.createDate(2005, 1, 1));
        reason1.setModificationDate(UnitTestHelper.createDate(2005, 2,
1));
        reason1.setCreationUser("TangentZ");
        reason1.setModificationUser("Ivern");
        reason1.setDescription(description1);
        reason1.setCompanyId(1);
        rejectReasons.put(new Long(reason1.getId()), reason1);

        // Create a manager for expense entry from configuration
        ExpenseEntryManager manager = new
ExpenseEntryManagerLocalDelegate();

        ExpenseEntry[] expected = new ExpenseEntry[5];

        // Add 5 instances
        for (int i = 0; i < 5; ++i) {
            expected[i] = new ExpenseEntry(i + 1);
            expected[i].setCompanyId(1);
            expected[i].setCreationUser("Create" + i);
            expected[i].setModificationUser("Modify" + i);
            expected[i].setDescription("Description" + i);
            expected[i].setAmount(new BigDecimal(i));
            expected[i].setBillable(true);
            expected[i].setDate(UnitTestHelper.createDate(2005, 2, i));
            expected[i].setExpenseType(type);
            expected[i].setStatus(status);
            expected[i].setRejectReasons(rejectReasons);
            manager.addEntry(expected[i], false);
        }

        // 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 = new
FieldMatchCriteria("expense_entry.expense_entry_id", new Integer(2));
        Criteria crit3 = new
FieldMatchCriteria("expense_entry.expense_type_id", new Integer(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(), new
Date());
        Criteria crit11 =
FieldBetweenCriteria.getCreationDateBetweenCriteria(new Date(), null);
        Criteria crit12 =
FieldBetweenCriteria.getCreationDateBetweenCriteria(null, new Date());
        Criteria crit13 =
FieldBetweenCriteria.getModificationDateBetweenCriteria(new Date(), new
Date());
        Criteria crit14 =
FieldBetweenCriteria.getModificationDateBetweenCriteria(new Date(),
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 = manager.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.