Time Tracker Common 3.1 Component Specification

1. Design

The Time Tracker Client custom component is part of the Time Tracker application. It provides an abstraction of clients that the projects are assigned to. This component handles the persistence and other business logic required by the application.

This design provides configurable ClientUtility which can be used to add, update and retrieve Client and associated Projects.

The clients can be search with filter and depth. Four kind of depth is provided. And pre-defined filter can be created by the ClientFilterFactory. The user can also create custom filter by the Filters provide by Search Builder component.

The action which will modify the database can be audited according to user's setting. And the audit will be rollback if the transaction is failed.

And the data source is pluggable, new data source can be easily added. And the alias of column name is provided by ClientColumnName and ClientProjectColumnName , the actual column name can be configured by the configuration file.

1.1 Design Patterns

- Data Access Object Pattern is used by ClientInformixDAO.
- Strategy Pattern is used to allow plugging different types of data source.
- Template Pattern is used to provide the general algorithm structure of Depth.
- Factory Pattern is used to create different filters by ClientFilterFactory.

1.2 Industry Standards

JavaBean
SQL

1.3 Required Algorithms

1.3.1 Constructs ClientUtility with given namespace

```
Get the configManager by ConfigManager.getInstance()
Get the objectFactoryNamespace from the configManager with namespace
and key as "ObjectFactoryNamespace"
Get the idName from the configManager with namespace and key
as "IDName"
Get idGenerator by IDGeneratorFactory.getIDGenerator(idName)
Create the ObjectFactory with a new ConfigManagerSpecificationFactory
created with the objectFactoryNamespace
Create dao by the ObjectFactory with the key as "ClientDAO"
Create auditManager by the auditManager with the key as "AuditManager",
if not specified, create with no-arg constructor.
Create addressManager by the addressManager with the key
as "AddressManager", if not specified, create with no-arg constructor.
Create contactManager by the contactManager with the key
as "ContactManager", if not specified, create with no-arg constructor.
Create commonManager by the commonManager with the key
as "CommonManager", if not specified, create with no-arg constructor.
Create projectUtility by the projectUtility with the key
as "ProjectUtility", if not specified, create with no-arg constructor.
```

1.3.2 *Constructs ClientInformixDAO with given namespace*

Get the configManager by ConfigManager.getInstance()
Get the connectionFactoryNamespace from the configManager with namespace and key as "ConnectionFactoryNamespace"
Create the connectionFactory by new DBConnectionFactoryImpl(connectionFactoryNamespace)
Get the connectionName from the configManager with namespace and key as "ConnectionName"
Get the searchBundleName from the configManager with namespace and key as "SearchBundleName"
Get the searchBundleNamespace from the configManager with namespace and key as "SearchBundleNamespace"
Create search bundle manager with searchBundleNamespace
Get searchBundle by manager.getSearchBundle(searchBundleName)
Get the Property containing all the column alias and actual name from the searchBundleNamespace. This property named "alias" contained by property named searchBundleName contained by property named "searchBundles".
Put all the alias, actual name pair into a HashMap. All the alias defined in ClientColumnNames should be found, and all the actual name should be non null, non empty(trim'd) string, else throw ConfigurationException.
Set the clientColumnNames to this map.
Get clientProjectColumnNames in the same way.

1.3.3 *Add client*

Throw InvalidClientPropertyException if:

- The company id <= 0
- Or description/name is null or empty(trim'd)
- Or the creation/modification user is null or empty(trim'd)
- Or the salesTax < 0
- Or the start/end Date, paymentTerm, contact or address is null

Set the ID of the client by idGenerator.getNextID()
If audit add client is true, create an auditHeader with all the properties of this client and createAuditRecord by AuditManager
Add the client by dao.addClient
  Set the creation and modification date to current date.
  Get a connection by createConnection, and prepare a statement to add the given client to the client table.
  Set all the parameters of statement according to the properties of client and executeUpdate
  Close the connection
Add the contact by contactManager.addContact(client.getContact()) and associate with this client by contactManager.associate
Add the address by addressManager.addAddress(client.getAddress()) and associate with this client by addressManager.associate
Call addProjectToClient to add the project
If add client failed and doAudit is true, call auditManager.delete(header.getID()) to rollback the audit.

1.3.4 *Add clients*

For each client ...
Throw InvalidClientPropertyException if any property of the contact is invalid.
Set the ID of the client by idGenerator.getNextID()
If audit add client is true, create an auditHeader with all the

properties of this client and create it by AuditManager
Add the id of the header to the auditId which is created by new long[client.length]
... end each
Call dao.addClients(clients)
  Get a connection by createConnection, and prepare a statement to add the given client to the client table.
  For each client, set the creation and modification date to current date and set all the parameters of statement according to the properties of client and addBatch.
  Execute the batch, and get the returned int[]
  If BatchUpdateException is thrown, get the int[] by exception.getUpdateCounts
  Create a boolean[], set boolean[i] to int[i]!=EXECUTE_FAILED
  If any int[i] is EXECUTE_FAILED, throw BatchOperationException which created with the boolean[]
  Close the connection
If BatchOperationException is thrown, get the boolean[] by exception.getResult
For each client which result[i]==true
  add the contact by contactManager.addContact(client.getContact()) and associate with this client by contactManager.associate
  add the address by addressManager.addAddress(client.getAddress()) and associate with this client by addressManager.associate
  Call addProjectToClient to add the project
For each client which result[i]==false, call auditManager.delete(auditId[i]) to rollback the audit.
If BatchOperationException have been thrown, rethrow it.

1.3.5 Update client

Throw InvalidClientPropertyException if:

- The company/client id <= 0
- Or description/name is null or empty(trim'd)
- Or the creation/modification user is null or empty(trim'd)
- Or the salesTax < 0
- Or the start/end Date, paymentTerm, contact or address is null

If the bean is not changed(client.isChanged() is false), simply return.
If audit update client is true, get the oldClient with the id by retrieveClient. Create an auditHeader with all the different properties between client and oldClient. Create audit by auditManager.create(header).
If the id of project of the client is changed, call removeProjectFromClient to deassociate with the old one and call addProjectToClient with the new one.
Simply call dao.updateClient(client)
  Set the modification date to current date.
  Get a connection by createConnection, and prepare a statement to update the client with given ID in the client table
  Set all the parameters of statement according to the properties of client and executeUpdate
  Close the connection
Update the contact by contactManager.updateContact(client.getContact()), if the contact id is changed, deassociate the oldClient.getContact with this client by contactManager.deassociate and associate the client.getContact with

this client by contactManager.associate
Update the address by addressManager.updateAddress(client.getAddress()), if the address id is changed, deassociate the oldClient.getAddress with this client by addressManager.deassociate and associate the client.getAddress with this client by addressManager.associate
If update client failed and doAudit is true, call auditManager.delete(header.getID()) to rollback the audit.

1.3.6 Update clients

Throw InvalidPropertyException if any property of any contact is invalid
Get all the ids of clients
Get the oldClients by retrieveClients(ids)?and get the result boolean[]
For each client...
If audit update client is true, create an auditHeader with all the different properties between client and oldClient. Create audit by auditManager.create(header).
Add the id of the header to the auditId which is created by new long[clients.length]
... end each
Call dao.updateClients() with all the clients with which isChanged is true.
  Get a connection by createConnection, and prepare a statement to update the client with given ID in the client table
  For each client, set the modification date to current date and create a sql statement to update the client and addBatch
  Execute the batch, and get the returned int[]
  If BatchUpdateException is thrown, get the int[] by exception.getUpdateCounts
  Create a boolean[], set boolean[i] to int[i]!=EXECUTE_FAILED
  If any int[i] is EXECUTE_FAILED, throw BatchOperationException which created with the boolean[]
  Close the connection
If BatchOperationException is thrown, get the boolean[] by exception.getResult
For each client which result[i]==true
  Update the contact by contactManager.updateContact(client.getContact()), if the contact id is changed, deassociate the oldClient.getContact with this client by contactManager.deassociate and associate the client.getContact with this client by contactManager.associate
  Update the address by addressManager.updateAddress(client.getAddress()), if the address id is changed, deassociate the oldClient.getAddress with this client by addressManager.deassociate and associate the client.getAddress with this client by addressManager.associate
  If the id of project of the client is changed, call removeProjectFromClient to deassociate with the old one and call addProjectToClient with the new one.
For each client which result[i]==false, call auditManager.delete(auditId[i]) to rollback the audit.
If BatchOperationException have been thrown, rethrow it.

1.3.7 Remove client

```
Get the client by retrieveClient(id)
If audit remove client is true, create an auditHeader with all the
properties of this client and create it by AuditManager
Call dao.removeClient(id)
  Get a connection by createConnection, and prepare a statement to
  remove the client with given id from the client table.
  Set the parameter of statement to given id and executeUpdate.
  Close the connection
Call removeProjectFromClient to deassociate with the project
Remove the contact and address of client by contactManager and
addressManager
For each project, remove project from the client.
deassociate the address with this client by
addressManager.deassociate
deassociate the contact with this client by
contactManager.deassociate
If remove client failed and doAudit is true, call
auditManager.delete(header.getID()) to rollback the audit.
```

1.3.8 Remove clients

```
Get the clients by retrieveClients(ids)
For each client ...
If audit remove client is true, create an auditHeader with all the
properties of this client and create it by AuditManager
Add the id of the header to the auditId which is created by new
long[clients.length]
... end each
Call dao.removeClients(ids)
  Get a connection by createConnection, and prepare a statement to
  remove the client with given id from the client table.
  For each client, set the parameter of statement to given id and
  addBatch
  Execute the batch, and get the returned int[]
  If BatchUpdateException is thrown, get the int[] by
  exception.getUpdateCounts
  Create a boolean[], set boolean[i] to int[i]!=EXECUTE_FAILED
  If any int[i] is EXECUTE_FAILED, throw BatchOperationException
  which created with the boolean[]
  Close the connection
If BatchOperationException is thrown, get the boolean[] by
exception.getResult
For each client which result[i]==true
  call removeProjectFromClient to deassociate with the project.
  Remove the contact and address by contact manager and address
  manager
  deassociate the address with this client by
  addressManager.deassociate
  deassociate the contact with this client by
  contactManager.deassociate
For each client which result[i]==false, call
auditManager.delete(auditId[i]) to rollback the audit.
If BatchOperationException have been thrown, rethrow it.
```

1.3.9 *Search client with filter and depth*

```
Get the result with dao.searchClients(filter, depth)
  Return searchBundle.search(filter, depth.getFields) which will
  return the result set including the fields specified by
  depth.getFields()
Get the clients by depth.buildResult(result)
  Get the number of records of the result set by result.getRecordCount
  Create a Client array with the number as size
  Set current row to the first row
  For each row
  Create a new Client by buildClient(Each specific depth will specify
  customized buildClient method)
  And put the Client into the array
  Goto next row
  ... end each
  Return the array.
If depth.setAddress is true, set address of each client by setAddress
If depth.setContact is true, set contact of each client by setContact
Set payment term by setPaymentTerm
If the depth.setProjects is true, if depth.onlyProjectsIdName() is
true, call setProjectsIdName, else call setProjects.
Return clients.
```

1.3.10 *Get all clients by DAO*

```
Get clients dao.getAllClients()
  Get a connection by createConnection, and prepare a statement to get
  all the clients from the client table
  Execute the query
  For each record
  Create a new Client according to the record, the payment term will
  be created only with the term id.
  And put the Client into a list
  ... end each
  Close the connection
  Return the clients in the list as Client[]
For each client ...
Set the address of the client
Set the contact of the client
Set the payment term of the client
Set the projects of the client
... end each
return clients
```

1.3.11 *Add project to client*

```
Call client.addProject(Project)
If audit add project to client is true, create an auditHeader with the
project id and client id and create audit by
auditManager.create(header)
Call dao.addProjectToClient
  Get a connection by createConnection, and prepare a statement to add
  the association to the client_project, the creation/modification
  user/date of association will be set to current date.
  Set all the parameters of statement according to the properties of
  the association and executeUpdate
  Close the connection
If add project is failed, call auditManager.delete(header.getId) to
```

```
rollback the audit, and remove the project by client.removeProject
```

1.3.12 Remove project from client

```
Call client.removeProject
If audit remove project to client is true, create an auditHeader with
the project id and client id and create audit by
auditManager.create(header)
Call dao.removeProjectFromClient
  Get a connection by createConnection, and prepare a statement to add
  the association with which clientId and projectID are equal to the
  given from client_project table
  Set all the parameters of statement according to the properties of
  the association and executeUpdate
  Close the connection
If remove project is failed, call auditManager.delete(header.getId)
to rollback the audit, and add the project by client.addProject
```

1.4 Component Class Overview

ClientUtility:

This class is the main class of this component. It provides various methods to retrieve and modify the client information. And it provides method to allow whether audit on the specified method. It can be configured with:

- Which implementation of ClientDAO will be used
- The name of ID Generator
- Construction of AuditManager, ContactManager, AddressManager, CommonManager and ProjectUtility

Client:

This class holds the information of a client.

ClientDAO:

This interface specifies the contract for implementations of a ClientDAOs. A ClientDAO is responsible for accessing the database. Clients' information is persisted in client table, while the associated project information is persisted in client_project table.

ClientInformixDAO:

This class is the Informix database implementation of the ClientDAO. It provides general retrieve/update/remove/add functionality to access the database. And it provides SearchClient method to search client with filter and depth. Clients' information is persisted in client table, while the associated project information is persisted in client_project table.

ClientColumnName:

This class is enumeration of the column name's alias in client table. It make all the column names can be configured.

ClientProjectColumnName:

This class is enumeration of the column name's alias in client_project table. It make all the column names can be configured.

Depth:

This abstract class is the base class of the depth. It defines the general interface for all the depth. Therefore, all depth depending search can be uniformly

processed without to know the difference between depths.

ClientOnlyDepth:
This class represents the client only depth. It extends Depth. Search with this depth, the result will contain only the client information.

ClientIDOnlyDepth:
This class represents the client ID only depth. It extends Depth. Search with this depth, the result will contain client name and Id only.

ClientProjectDepth:
This class represents the client project depth. It extends Depth. Search with this depth, the result will contain all clients and all projects.

SummaryDepth:
This class represents the summary depth. It extends Depth. Search with this depth, the result will contain client id & Name and associated project ids & names.

ClientFilterFactory:
This class is used to create pre-defined filters or AND/OR/NOT filter based on given filters.

1.5 Usage of Exceptions

ClientUtilityException:
This exception will be the base exception for all exceptions thrown by the ClientUtility. This exception can be used by the application to simply their exception processing by catching a single exception regardless of the actual subclass.

InvalidClientIDException:
This exception will be created and thrown by the ClientUtility when it can't generate ID successfully. This exception will be exposed to the caller of ClientUtility.addClient(s) method.

InvalidClientPropertyException:
This exception will be thrown by the ClientUtility when the property of the properties of given Client is invalid. This exception will be exposed to the caller of ClientUtility's addClient(s) and updateClient(s) method.

PropertyOperationException:
This exception will be thrown by the ClientUtility and ClientDAO if it encounters any exception when try to retrieve/delete/update/create the contact/address/project property from contact/address manager or project utility. It will be exposed to the caller of the batch operation methods.

ClientPersistenceException:
This exception will be thrown by the ClientUtiltiy and the implementations of ClientDAO when they encounter database exceptions. This exception will be exposed to the caller of ClientUtiltiy and the ClientDAO implementations' database related methods.

ConfigurationException:
This exception will be thrown by the ClientUtiltiy and the implementations of ClientDAO when they encounter configuration error. This exception will be exposed to the caller of constructor of ClientUtility and ClientDAO implementations.

ClientAuditException:
This exception will be thrown by the ClientUtility if it encounters exception when audit. This

exception will be exposed to the caller of ClientUtility's addClient(s), updateClient(s), removeClient(s), addProjectToClient and removeProjectFromClient method.

BatchOperationException:
This exception will be thrown by the ClientUtility if the batch operation can't be completed successfully. This exception will be exposed to the caller of ClientUtility's batch operation methods.

1.6 Thread Safety

Client is not thread-safe by being mutable. They are not supposed to be used in multithread environment. If they would be used in multithread environment, they should be synchronized externally.

Other classes in this component are thread-safe by being immutable.

2. Environment Requirements

2.1 Environment

- Development language: Java 1.4
- Compile target: Java 1.4, Java 1.5

2.2 TopCoder Software Components

- Configuration Manager 2.1.5 is used to provide configuration options
- Object Factory 2.0 is utilized to create the implementations of ClientDAO.
- Base Exception 1.0 is used to provide a base for all custom exceptions.
- ID Generator 3.0 is used to generate IDs for Client.
- DB Connection Factory 1.0 is used to generate the database Connection.
- Type Safe Enum 1.0 is used to be extended by the ClientColumnName and ClientProjectColumnName enumeration.
- Search Builder 1.3.1 is used to enable search with filter
- Database Abstraction 1.1 is used to provide the CustomResultSet return by the SearchBundle
- Time Tracker Audit 3.1 is used to provide audit functionality
- Time Tracker Common 3.1 provides TimeTrakerBean base class, PaymentTerm and CommonManager
- Time Tracker Project 3.1 provides Project and ProjectUtility to get Projects.
- Time Tracker Contact 3.1 provides Contact and Address and their manager

2.3 Third Party Components

None

3. Installation and Configuration

3.1 Package Name

com.topcoder.timetracker.client

com.topcoder.timetracker.client.db

com.topcoder.timetracker.client.depth

3.2 Configuration Parameters

3.2.1 Configuration Parameters

ClientUtility has the following configuration using the specified namespaces (or one provided by the application):

Namespace: com.topcoder.timetracker.client.ClientUtility

Property Name	Description	Format	Required
`objectFactoryNamespace`	The namespace used for ObjectFactory to create the `ClientDAO`	String	Yes
`IDName`	The name of ID Generator	String	Yes

The ClientUtility will use the ObjectFactory to create the following:

Class Type	Key	Required
com.topcoder.timetracker.client.`ClientDAO`	`"ClientDAO"`	`Yes`
com.topcoder.timetracker.contact.ContactManager	`"ContactManager"`	`No`
com.topcoder.timetracker.contact.AddressManager	`"AddressManager"`	`No`
com.topcoder.timetracker.common.CommonManager	`"CommonManager"`	`No`
com.topcoder.timetracker.project.ProjectUtility	`"ProjectUtility"`	`No`

The ClientInformixDAO has the following configuration using the specified namespaces (or one provided by the application or ClientUtility):

Namespace: com.topcoder.timetracker.client.db.ClientInformixDAO

Property Name	Description	Format	Required
`ConnectionFactoryNamespace`	The namespace used to create DBConnectionFactory	String	Yes
`ConnectionName`	The name used to get connection from connection factory	String	No
`SearchBundleName`	The name will be used to get SearchBundle from SearchBundleManager	String	Yes
`SearchBundleNamespace`	The namespace used to create `SearchBundleManager`	String	Yes

NOTE: In the configuration of SearchBundleNamespace's alias property, should specify all the aliases defined in ClientColumnName and ClientProjectColumnName. Please refer to sample configuration file.

3.2.2 Sample Configuration

Please refer to config.xml

3.3 Dependencies Configuration

None

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute 'ant test' within the directory that the distribution was extracted to.

4.2 Required steps to use the component

Nothing special required

4.3 Demo

4.3.1 Create ClientUtility

```
//Create a new ClientUtility, the default namespace will be used
ClientUtility utility = new ClientUtility();

// Create a new ClientUtility with namespace
ClientUtility utility = new ClientUtility("com.topcoder.timetracker.client");
```

4.3.2 Add Client(s)

```
//Create a new Client
Client client = new Client();

//Set the properties
client.setCreationUser("CreationUser");
client.setModificationUser("ModificationUser");
client.setDescription("This is description");
client.setTerm(3);
client.setActive(true);
client.setChanged(true);

//Add the client by utility, and this action will be audited
utility.addClient(client, true);

//Add the clients by utility, clients is Client[], and this action will be
audited
utility.addClients(clients, true);
```

4.3.3 Update Client(s)

```
//update the client by utility, and this action will be audited
utility.updateClient(client, true);

//update the clients by utility, clients is Client[], and this action won't
be audited
utility.updateClients(clients, false);
```

4.3.4 Retrieve Client(s) by ID

```
//retrieve the client by utility
Client client = utility.retrieveClient(clientId);
```

```
// retrieve the clients by utility, clientIds is long[]
Client[] clients = utility.retrieveClients(clientIds);
```

4.3.5 Remove Client(s) by ID

```
//remove the client by utility, and this action won't be audited
utility.removeClient(clientId, false);

//remove the clients by utility, clientIds is long[], and this action will
be audited
utility.removeClients(clientIds, true);
```

4.3.6 Get all clients

```
//get all clients by utility
Client[] clients = utility.getAllClients();
```

4.3.7 Search clients with filter and depth

```
//Search with filter (About creating filters, please refer Create various filters)
and client id only depth.
Client[] clients = utility.searchClient(filter, new ClientIDOnlyDepth());
```

4.3.8 Add project to client

```
//add project to client, and this action will be audited
utility.addProjectToClient(clientId, projectToBeAdded, true);
```

4.3.9 Remove project from client

```
//remove project from client, and this action won't be audited
utility.addProjectFromClient(clientId, projected, false);
```

4.3.10 Get all projects of client

```
//get all projects of a client
Project[] projects = utility.getAllProjectsOfClient(clientId);
```

4.3.11 Create various filters

```
//Create a company id filter which will return all clients with a given company
ID
Filter companyIdFilter =
ClientFilterFactory.createCompanyIdFilter(companyId);

//Create a name keyword filter which will return all clients with name that
contains a given string, the keyword is a string should be included in the
name
Filter nameKeywordFilter =
ClientFilterFactory.createNameKeywordFilter(keyword);

//Create a created in filter which will return all clients created within a
given inclusive date range (may be open-ended). To can be null, which indicate
```

```
the range is open-ended.
Filter createdInFilter = ClientFilterFactory.createCreatedInFilter(from,
to);

//Create a modified in filter which will return all clients modified within
a given inclusive date range (may be open-ended) . To can be null, which
indicate the range is open-ended.
Filter modifiedInFilter = ClientFilterFactory.createModifiedInFilter(from,
to);

//Create a created by filter which will return all clients created by a given
username.
Filter createdByFilter =
ClientFilterFactory.createCreatedByFilter(userName);

// Create a modified by filter which will return all clients modified by a
given username.
Filter modifiedByFilter =
ClientFilterFactory.createModifiedByFilter(userName);

// Create an AND filter with created by filter and company id filter which
will return all clients created by a given username and company id
Filter andFilter = ClientFilterFactory.andFilter(createdByFilter,
companyIdFilter);

// Create an OR filter with two created by filter which will return all clients
created by the two given usernames
Filter orFilter = ClientFilterFactory.andFilter(createdByFilter,
createdByFilter2);

// Create a NOT filter with created in filter which will return all clients
not created in the given date range
Filter notFilter = ClientFilterFactory.notFilter(createdInFilter);
```

5. Future Enhancements

Implement accessibility for other database systems.